Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

AI Scientists for Building Virtual-Cell Models

Mark Gerstein

pi@gersteinlab.org

Yale University https://orcid.org/0000-0002-9746-3719

Xiangru Tang

Yale University https://orcid.org/0009-0006-2700-4513

Jiapeng Chen

Yale University

Yanjun Shao

Yale University https://orcid.org/0009-0005-3486-467X

Zhuoyun Yu

Stanford University

Yan Cui

University of Pennsylvania

Weixu Wang

Technical University of Munich

Fang Wu

Stanford University https://orcid.org/0000-0001-7240-3915

Yuxuan Liao

Yale University

Yuchen Zhuang

Georgia Institute of Technology

Wenqi Shi

Washington University in St. Louis

Zhi Huang

University of Pennsylvania https://orcid.org/0000-0001-6982-8285

Arman Cohan

Allen Institute https://orcid.org/0000-0002-8954-2724

Xihong Lin

Harvard T.H. Chan School of Public Health https://orcid.org/0000-0001-7067-7752

Smita Krishnaswamy

Yale School of Medicine https://orcid.org/0000-0001-5823-1985

Fabian Theis

Helmholtz Munich https://orcid.org/0000-0002-2419-1943

Biological Sciences - Article

Keywords: virtual cell modeling, perturbation prediction, single-cell genomics, autonomous scientific discovery

Posted Date: September 17th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10954762/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: There is **NO** Competing Interest.

AI Scientists for Building Virtual-Cell Models

Xiangru Tang[1*†], Jiapeng Chen[2,5†], Yanjun Shao[2,4,5†], Zhuoyun Yu[1†], Yan Cui[8], Weixu Wang[6,7], Fang Wu[10], Yuxuan Liao[2], Yuchen Zhuang[11], Wenqi Shi[12], Zhi Huang[8,9], Arman Cohan[2], Xihong Lin[13], Smita Krishnaswamy[2], Fabian Theis[6,7], Mark Gerstein[1,2,3,4,5*]

[1]Department of Computer Science, Yale University, New Haven, Connecticut, USA.
[2]Program in Computational Biology and Biomedical Informatics, Yale University, New Haven, Connecticut, USA.
[3]Department of Molecular Biophysics and Biochemistry, Yale University, New Haven, Connecticut, USA.
[4]Department of Statistics and Data Science, Yale University, New Haven, Connecticut, USA.
[5]Department of Biomedical Informatics and Data Science, Yale School of Medicine, Yale University, New Haven, Connecticut, USA.
[6]Institute of Computational Biology, Helmholtz Munich, Munich, Germany.
[7]School of Computation, Information and Technology, Technical University of Munich, Munich, Germany.
[8]Department of Bioengineering, University of Pennsylvania, Philadelphia, Pennsylvania, USA.
[9]Department of Pathology and Laboratory Medicine, Perelman School of Medicine, University of Pennsylvania, Philadelphia, Pennsylvania, USA.
[10]Department of Computer Science, Stanford University, Stanford, California, USA.
[11]Department of Computer Science Engineering, Georgia Institute of Technology, Atlanta, Georgia, USA.
[12]Department of Genetics, Washington University in St. Louis, St. Louis, Missouri, USA.
[13]Department of Biostatistics, Harvard T.H. Chan School of Public Health, Boston, Massachusetts, USA.

*Corresponding author(s). E-mail(s): xiangru.tang@yale.edu; mark.gerstein@yale.edu;
†These authors contributed equally to this work.

Abstract

Autonomous agents, from virtual laboratories to biological data-analysis agents and co-scientists, have begun to turn scientific goals into hypotheses. Whether they can go further and convert real, noisy and complex biological data into executable model hypotheses and architectures that encode testable assumptions remains unclear. Predicting how cells respond to genetic and chemical perturbations (i.e., modeling virtual cells) is a natural test of this ability: it is central to decoding disease mechanisms and therapeutic discovery, yet it demands cross-disciplinary expertise rarely concentrated in a single research group. Here, we introduce CellForge, an organized team of autonomous agents for single-cell perturbation modeling. Given a dataset and a task, it surveys the literature, parses the data, identifies relevant baselines, and formulates a model hypothesis through critic-mediated discussion among agents specialized in data modeling, single-cell biology, and

deep learning. It then writes the training code, runs the experiments, and returns a trained predictor including architecture, code, and validation results. The architecture itself is the hypothesis: CellForge produces it end-to-end, with no human specifying the design, loss, or training protocol. Across six perturbation datasets spanning transcriptomic, epigenomic, and proteomic contexts, CellForge yields models that are competitive with hand-crafted state-of-the-art results on established tasks and extends the same workflow to paired multi-omic readouts. Blinded human evaluators rate CellForge's research plans as more scientifically sound and innovative than those from single LLMs or deep-research systems. In summary, we show autonomous agents can complete the full hypothesis-to-validation cycle in single-cell biology, with an AI effectively building another AI, in a complex setting that currently requires a team of human experts.

Keywords: virtual cell modeling, perturbation prediction, single-cell genomics, autonomous scientific discovery

Introduction

Modern biological research increasingly relies on high-dimensional perturbation datasets that span multiple intervention types and molecular readouts. Building a virtual cell that can predict cellular responses across these perturbations is an important objective for computational biology [1] and has been formalized in the Virtual Cell Challenge [2]. In parallel, the field is moving toward multimodal representation learning across transcriptomic, epigenomic, proteomic, and phenotypic measurements [3]. Despite this progress, reliable generalization across unseen perturbation contexts remains limited, and the practical question of how to design transferable perturbation models is still unresolved.

The community has developed a rich toolkit for perturbation prediction, including conditional variational models [4–7], optimal transport [8, 9], interactome-informed graph networks [10], flow-based models [11], and transformers [12, 13], each demonstrating value on particular benchmarks. Yet no architectural choice transfers reliably across perturbation types, readout modalities, or evaluation protocols: performance claims depend strongly on split design and dataset composition [14–19], and deep models do not consistently outperform linear baselines under matched evaluation [20]. Experimental scope has meanwhile expanded from transcriptomics to chromatin accessibility, paired RNA-protein, cytokine dynamics, and spatial context [21–25], spanning genetic screens, chemical treatments, and cytokine responses across increasingly large atlases [26–32], with cross-context and cross-modality modeling still maturing [33–35]. Each new setting has required rebuilding models from scratch, drawing on expertise rarely concentrated in a single group: the bottleneck is no longer data or compute, but the sustained effort of model design. A model hypothesis in this setting is an executable design: an architecture, loss, and training protocol that encode testable assumptions about cellular response. Here, we use a team of autonomous agents to mean a coordinated set of large language model (LLM)-driven agents, each assigned a scientific or engineering role, that retrieves evidence, calls software tools, critiques proposals, and revises a shared research product with limited human intervention [36–38]. Together, these trends define the challenge for an autonomous agent in perturbation biology: the system must read the literature, formulate a dataset-tailored model hypothesis, implement it, and evaluate it against well-accepted, community-defined splits. The relevant test is end-to-end completion in single-cell biology, rather than automation of isolated steps.

This challenge goes beyond any single architecture. It asks whether scientific reasoning about the model space itself can be automated: whether a system can formulate hypotheses grounded in biological literature, adapt them to each dataset, and instantiate them without manual design. Recent work shows that AI can assist parts of this cycle, from literature synthesis and hypothesis generation [39–43] to autonomous software development [44–48], with early applications in biomedicine spanning gene-set analysis, experiment automation, and single-cell exploration [37, 38, 49–52]. Most recently, end-to-end autonomous AI research systems have demonstrated automated idea generation, code execution, and empirical evaluation in machine-learning settings [53]. These systems clarify the operational template for an autonomous agent, but their benchmarks largely focus on self-contained computational tasks. Single-cell perturbation prediction is a sharper scientific testbed: each dataset requires data processing choices, single-cell biological assumptions, and deep-learning design to be reasoned about jointly, and an error in any one of them can invalidate an otherwise plausible

architecture. Calls to extend autonomous AI research to such scientific domains [36, 39, 54] have therefore outpaced concrete demonstrations. Existing systems still leave this gap on both sides: systems that reason over biology have not produced executable models [40, 51], whereas systems that generate and run code have not grounded their designs in biological deliberation [44–46]. The full reasoning cycle from scientific hypothesis to trained, benchmarked model remains open, particularly for perturbation biology where model design choices interact tightly with evaluation protocol [15].

Here we present CELLFORGE, a team of autonomous agents for perturbation model development. To our knowledge, CELLFORGE is the first system to close the design-to-implementation loop in this setting. Given a single-cell perturbation dataset and a task specification, its agents parse the data, benchmark existing baselines, survey the relevant literature, formulate a model hypothesis through structured multi-agent discussion, write and execute the training code, and return an executable model hypothesis: trained weights and an architecture specification, together with code and validation results. In this formulation, the model architecture itself is the hypothesis, proposed and validated end-to-end without human design intervention. We evaluate CELLFORGE on six single-cell perturbation datasets (seven held-out prediction readouts) spanning genetic, chemical, and cytokine perturbations across scRNA-seq, scATAC-seq, and CITE-seq, asking whether autonomous hypothesis generation can substitute for manual model design across diverse perturbation settings. Crucially, the evaluation is fixed before model generation begins: tasks, held-out splits, and metrics follow community-defined criteria, so CELLFORGE's generated models are judged against standards external to the system rather than against protocols it selected or graded itself.

Results

CellForge turns a biological task into an executable model hypothesis

Predicting how cells respond to a perturbation has become a central computational problem in single-cell biology, yet every new perturbation atlas currently demands a bespoke predictor, designed by a team that spans cell biology, statistics, and deep learning. The first test for CellForge is therefore whether it can replace this manual model-design step by producing, from a concrete perturbation-prediction task, not only a written proposal but an executable model hypothesis (Figure 1a,b). Given a single-cell perturbation dataset and a task description, CellForge returns a trained model together with its architecture specification and training code. In this setting, the model hypothesis is the executable predictor itself: its architecture, loss, and training protocol encode assumptions about how perturbations change cell state. We test these assumptions across the perturbation classes and molecular readouts summarized in Figure 1c, treating candidate model architectures as hypotheses to be selected and evaluated empirically (Figure 1d).

CellForge turns the task into this executable hypothesis through three stages (Figure 2a). Formulation fixes the empirical problem before any new architecture is proposed; then during Ideation, expert proposals undergo critic-mediated review and revision; the selected research plan then passes to coding agents for iterative implementation and execution (Figure 2b). In more detail, (1) **Formulation** parses the perturbation, feature, and context fields; constructs held-out perturbation splits; reproduces established baselines from previous papers; and locks the metric pipeline into a task report. (2) **Ideation** uses this task report as the basis for model design. Data modeling, single-cell biology, and deep learning agents retrieve literature, propose candidate designs, critique one another's assumptions, and revise the preprocessing strategy, neural architecture, loss function, and training protocol through critic-mediated discussion, with coordination trajectories tracked across rounds (Figure 2c). The protocol mirrors conference peer review: domain experts first-author separate proposals and then review one another's plans. A central critic acts like an area chair, synthesizing the reviews into a meta-review that guides revisions and final selection. (3) **Execution** treats the approved plan as an implementation target: it writes runnable training code, launches five-fold cross-validation, repairs failed implementations, and returns final model weights, an architecture specification, source code, and validation metrics.

We applied the same formulation–ideation–execution loop to six perturbation-response datasets spanning scRNA-seq, paired RNA-protein CITE-seq, and scATAC-seq. These experiments test whether CellForge can design models that predict cell responses to unseen perturbations, a setting where generalization has traditionally been difficult. The remainder of the Results therefore proceeds in three steps: first, we summarize

the architectural motifs selected by CellForge; next, we test performance on established scRNA-seq benchmarks with published reference methods; and finally, we evaluate under-studied perturbation settings where task-specific predictors are limited.

Before comparing predictive performance, we first inspected whether generated architectures varied with the biological task. Across tasks, the architectural motifs selected by CellForge exhibited clear task-specific patterns rather than converging on a single template. Among the established scRNA-seq benchmarks, single-gene CRISPRi models primarily used MLP and attention components, combinatorial CRISPRa models more often included attention and bottleneck components, and drug-response models showed the widest range of motifs, including VAE-style latent encoders, optimal-transport modules, and normalizing-flow components. The under-studied settings extended this pattern: temporal cytokine models used VAE and MLP-mixing components suited to continuous response trajectories; CITE-seq models used graph-attention and Transformer components to combine paired RNA and protein readouts; and scATAC-seq models used diffusion-based denoising modules suited to sparse, near-binary chromatin accessibility data. The next two sections examine these dataset-specific architectures in detail and evaluate their empirical performance.

CellForge designs competitive architectures on established scRNA-seq benchmarks

CellForge did not converge on a single universal architecture. Instead, the generated models showed task-specific patterns that often aligned with the structure of each perturbation problem. We first considered three established scRNA-seq benchmarks for which published reference methods are available: single-gene CRISPRi [26], combinatorial CRISPRa [27], and drug perturbation [28]. For genetic perturbations, the published references included CPA, scGen, CondOT, Biolord and scGPT [5, 6, 8, 55, 56]. For drug perturbations, they included ChemCPA and CellFlow [11, 57], alongside unperturbed, random-forest and linear-regression controls. We reproduced the applicable methods and evaluated them with CellForge using the same five held-out-perturbation folds and metric pipeline (Methods). Figure 3a–c illustrates the three CellForge architectures, with full implementation details provided in Supplementary Section A.1. Figure 3d summarizes global and differential-feature prediction, while panels e and f assess differential-feature recovery through receiver operating characteristic (ROC) curves and the corresponding area under the ROC curve (AUROC), and panel g aggregates ranks across the same metric families. Differential-feature recovery tests whether features ranked by absolute predicted change recover the top 50 features in the observed perturbation-versus-control comparison (Methods).

For single-gene CRISPRi, CellForge proposed *CPA-X*, an MLP expression encoder coupled to a multi-head cross-attention block in which a perturbation embedding attends to the encoded control-cell state. This design is related to ideas in scGen and CPA [5, 55], and is well matched to a setting where each example is defined by one genetic perturbation and one observed expression response. Empirically, CPA-X achieved the highest Pearson correlation on the Adamson benchmark, indicating strong recovery of global expression structure. Its differential-expression error was higher than that of the best reference methods, suggesting that global expression agreement and differential-gene fidelity remain partly separable objectives.

For combinatorial CRISPRa, CellForge generated *scGen-X*, a bottlenecked expression encoder with a cross-attention module that fuses perturbation identity with cell context before a residual MLP decoder. This architecture resembles design principles used in GEARS and AttentionPert [58, 59], and is consistent with the need to model interactions between paired perturbations. On the Norman benchmark, scGen-X achieved the lowest differential-expression mean squared error, showing that the generated design was competitive in a setting where combinatorial effects are central to the task.

For drug response, CellForge generated *ChemCellFlow*, which combines gene and chemical encoders, conditions on compound identity and dose, applies a Sinkhorn-based conditional optimal-transport block, and refines the latent state with a normalizing-flow module. This architecture is closest to CellFlow and related neural optimal-transport approaches [9, 11, 57], suggesting that CellForge recovered the main design principle of a strong human-designed model: treating drug response as a conditional transport problem between cellular states. On the Srivatsan drug benchmark, CellFlow [11], which was developed and tuned specifically for this setting, achieved the best performance. ChemCellFlow achieved the second-best result, outperforming

the remaining reference methods despite using a simpler coupling of chemical, dose, and cellular state representations. Thus, even when the best method was a specialized human-designed model, CellForge identified the relevant architecture family and automatically produced a competitive implementation.

Together, these results show that CellForge can recover strong architecture families for established scRNA-seq perturbation benchmarks, with performance competitive with published reference methods while preserving task-specific model structure.

CellForge extends model design to under-studied perturbation settings

We next asked whether CellForge could design models beyond the established expression-only endpoint benchmarks above. We considered three settings where task-specific reference predictors are limited: cytokine-driven differentiation dynamics in the Schiebinger dataset [60], paired RNA-protein perturbation readouts in the Papalexi CITE-seq dataset [22], and chromatin-accessibility perturbation in the Liscovitch-Brauer scATAC-seq dataset [21]. These tasks test whether an autonomous design system can adapt to time-dependent responses, multimodal readouts, and sparse regulatory measurements. To our knowledge, no manually crafted, published task-specific prediction models are available for these datasets. We therefore benchmarked CellForge against a common reference set of four established model types (linear regression, random forest, graph neural network and Transformer), all trained and evaluated under an identical protocol. Figure 4a–c illustrates the representative CITE-seq, scATAC-seq, and cytokine architectures, respectively, with full implementation details provided in Supplementary Section A.1.

For cytokine-driven differentiation, CellForge generated *CPA-Traj*, a trajectory-aware VAE with shared and condition-specific latents, a continuous time covariate, and a perturbation-mixing module before an MLP decoder. This design reflects a setting where perturbation effects unfold along a continuous response trajectory rather than as a single endpoint. CPA-Traj ranked first on several global and differential-feature metrics (Figure 4d). Differential-feature recovery is reported using ROC curves and AUROC in Figure 4e,f, respectively.

For paired RNA-protein CITE-seq, CellForge used the joint architecture *totalGAT* shown in Figure 4a. The title-case label *TotalGAT* in panel a denotes the same model. We refer to it as *totalGAT* throughout the text. *totalGAT* used modality-specific encoders, graph-attention layers, cross-attention with perturbation embeddings, and separate output heads for RNA and protein. This design reflects the need to combine two molecular readouts while preserving distinct output spaces. On the Papalexi dataset, CellForge achieved the lowest MSE for both RNA and protein prediction and improved RNA differential-expression correlation relative to the generic baselines, although generic neural baselines remained competitive on some global correlation metrics. Figure 4d shows global and differential-feature prediction for both CITE-seq readouts, with differential-feature recovery summarized by ROC curves and AUROC in panels e and f and joint rankings in panel g. Recovery was weaker for protein than for RNA, consistent with the difficulty of predicting perturbation-sensitive protein features. This pattern indicates partial recovery of paired-modality structure rather than uniformly strong performance across both molecular readouts.

For scATAC-seq, CellForge generated *ChromDDPM*, a DDPM-style [61] denoising model conditioned on cell state and perturbation information. This design is a natural fit for sparse, near-binary chromatin-accessibility profiles, where prediction can be viewed as recovering perturbation-sensitive accessibility signal from a noisy high-dimensional readout. On the Liscovitch-Brauer dataset, overall performance was lower than in the expression-based tasks, but CellForge improved over simple baselines on several metrics (Figure 4d,e,g). At the same time, the modest global performance and recovery curves indicate partial rather than complete recovery of regulatory response structure.

Together, these results show that CellForge can extend beyond well-studied scRNA-seq benchmarks. In settings where task-specific perturbation predictors are scarce, it assembled established machine-learning components into modality-aware architectures that were often competitive with generic neural baselines and generally stronger than simple learners, without manual architecture engineering.

Blinded evaluators prefer CellForge's research plans over single-agent and deep-research alternatives

The benchmark results above evaluate final trained models, but they do not by themselves show whether CellForge improves the quality of the research plans that precede code execution. To test this directly, we compared CellForge plans with plans generated from identical task specifications by OpenAI Deep Research [62], Perplexity Deep Research [63], Gemini Deep Research [64], Biomni [38], and a single-LLM prompting baseline. All plans were evaluated blind to system identity. An example trace shows the conference-review-like process used to generate a CellForge plan: experts propose and review plans, and an AC-like critic provides a meta-review that guides revision (Figure 5a).

In the LLM-judge evaluation, CellForge achieved the highest mean score in 14 of 15 task–criterion combinations (Figure 5b); OpenAI Deep Research scored higher only for technical feasibility on the drug task. A separate blinded evaluation by three single-cell biology experts assessed technical novelty, feasibility, clarity and consistency, biological plausibility, mechanism explanation, pathway relevance, and cross-perturbation generalizability (Figure 5c). This human evaluation likewise favored CellForge across most task–criterion combinations, although not uniformly. Pairwise correlations among the five LLM judges ranged from 0.88 to 0.93, indicating strong agreement among their plan evaluations (Figure 5d). Mean human-expert scores correlated more strongly with the corresponding LLM-judge scores than with CellForge's internal graph-based discussion confidence score, indicating that the external LLM judges more closely tracked human assessments than did the system's internal deliberation signal (Figure 5e). Phase-resolved profiles further show how the systems were judged on task analysis and method design across the three perturbation settings (Figure 5f). Together, these complementary LLM and human reviews provide plan-level evidence that the multi-agent deliberation improved judged plan quality before code execution. The human panel was small, however, and neither evaluation constitutes independent biological validation or evidence about final trained-model performance.

Discussion

CellForge demonstrates that model design itself can be treated as a scientific hypothesis: rather than selecting from predefined architectures, an autonomous team reasons over the literature, formulates dataset-tailored modeling hypotheses, and instantiates them as trained, benchmarked models. The resulting architectures were synthesized through scientific reasoning conditioned on each dataset's characteristics rather than drawn from a fixed search space, and produced competitive predictions across genetic, chemical, and cytokine perturbation benchmarks, including the highest Pearson correlation on Adamson CRISPRi ($PCC = 0.9883$) and the lowest differential-expression error on Norman CRISPRa ($MSE_{\mathrm{DE}} = 0.1736$). Prior agentic systems for science occupy distinct, narrower roles. Virtual Lab [37] orchestrates existing tools such as ESM and Rosetta into pipelines but does not invent architectures. CellVoyager [51] generates hypotheses from biological data but stops short of implementation. *The AI Scientist* [53] automates template-based ML ideation and coding but evaluates by self-review and is not grounded in domain literature. CellForge closes this gap by treating model architecture itself as a hypothesis to be reasoned over, implemented, and evaluated against fixed, community-defined benchmark splits that precede model design, an inversion that addresses the reproducibility problem in computational biology where evaluation protocol and model development are otherwise entangled. A further distinction is that *The AI Scientist* validates outputs through LLM-generated self-review, whereas CellForge validates against held-out perturbation conditions with experimentally measured single-cell readouts, a requirement that is not negotiable in the single-cell setting. Together these properties establish that autonomous research can produce computational methods validated on the same held-out perturbation splits as the human-designed methods they compete against.

While CellForge produces competitive perturbation models, the current system has several limitations. Three runs per dataset is too few to characterize the full distribution of architectures CellForge would generate across many runs. More broadly, recent evidence suggests that AI tools expand individual scientific impact while contracting science's collective focus [65]; autonomous systems like CellForge may be subject to the same dynamic, as LLM-driven ideation can systematically favor architectures that are well-represented in training data regardless of dataset-specific conditioning, and whether the critic-mediated deliberation genuinely

diversifies architectural choices across datasets has not been directly measured. Human review is intentionally minimal, limited to two steps: a brief review of research plan plausibility before execution begins, and training-script compliance before any GPU job is launched, to prevent harmful or resource-wasteful execution. Whether this two-step design represents the right minimal footprint and whether it generalizes to more complex task specifications have not been tested directly. Validation here is also computational: CellForge's predictions are scored against held-out perturbations measured experimentally by the original authors, but the framework has not yet been coupled to a wet-lab loop, and prospective experimental validation of its predictions remains future work. Spatially resolved perturbation transcriptomics is a natural next testbed for the framework itself. Such datasets replace unordered cell sets with spatial graphs and may require different model classes [25, 66, 67], and no dominant modeling paradigm has yet emerged for them.

The broader significance of CellForge is that it attempts to design models, not just assemble pipelines or analyze data. In this setting, model architecture is itself a scientific hypothesis. Choices about the encoder reflect assumptions about how perturbation signals move through gene regulatory space, whereas choices about the decoder reflect assumptions about the structure of gene expression manifolds. The next question is whether the agents' explanations for these choices match the mechanisms that actually drive model performance. Figure 5 shows that blinded LLM judges and a separate panel of single-cell biology experts generally preferred CellForge's research plans. The strong correspondence between human and LLM scores supports the use of LLM judges for plan-level comparison, whereas their weaker correspondence with CellForge's internal confidence score indicates that deliberation confidence is not itself a substitute for external evaluation. The agents reason meaningfully about the task, but stronger tests will be needed to determine whether that reasoning is mechanistically faithful. Perturbation prediction is a useful proving ground because it is both well benchmarked and a rapidly advancing area of single-cell machine learning, making proposed designs straightforward to compare against strong baselines. More broadly, autonomous method design may prove most valuable in ill-defined biological settings, where no dominant modeling paradigm yet exists and where a human team would otherwise spend months on architecture exploration before reaching a working baseline.

Methods

CELLFORGE takes as input a processed single-cell perturbation dataset and a task description; the pipeline is grounded by a community-agreed evaluation criterion fixed before any architecture is proposed and a seed corpus of key papers used to initialize literature retrieval. It returns a trained model with its design (model weights, architecture specification, and training code). A team of autonomous agents parses the data, benchmarks existing baselines, retrieves additional literature, formulates a model hypothesis through multi-agent discussion, generates code, trains the model, and reports results, across three sequential stages: Formulation, Ideation, and Execution (Figure 2). Human review is restricted to two steps: research plan plausibility before execution begins, and training-script compliance before any GPU job is launched. Full prompt templates and run logs are released with the codebase.

Overview

Problem formulation.

We study held-out perturbation prediction. For a control-state molecular profile x, perturbation identity p, and optional context c (for example cell type, batch, or donor), the goal is to learn a predictor $f_\theta(x, p, c) \mapsto y$ that returns the post-perturbation molecular profile y for perturbations not observed during training. Prediction is typically mediated by a learned encoder $g_\phi(x) = z$ that maps cells into a latent space before applying perturbation-specific transformations; however, the encoder, decoder, and loss are designed per dataset rather than fixed in advance. Perturbation identities are split into five folds, and we evaluate all methods only on held-out perturbations. We score all baselines and CellForge-designed models under the same split definition and metric pipeline. For deterministic baselines, the reported score is the average over folds. Each CellForge run comprises a full design-implement-train cycle evaluated across the same five folds. Summary values reported in the manuscript are averaged over three independent runs.

Evaluation metrics.

Following recent perturbation benchmarks and evaluation tooling [13, 58, 68], metric computation is perturbation-centric: scores are computed per held-out perturbation rather than per cell. For each held-out perturbation (and matched context, when context labels are available), we first aggregate predicted cells and observed cells separately into mean profiles over the shared feature space. We then compute mean squared error (MSE), Pearson correlation (PCC), and the coefficient of determination ($R^2 = 1 - \mathrm{SS}_{\mathrm{res}}/\mathrm{SS}_{\mathrm{tot}}$, which can be negative when predictions are worse than the mean) between the predicted and observed profiles across features for that perturbation. For visualization, MSE is transformed as $-\log_{10}(\max(\mathrm{MSE}, 10^{-12}))$, so larger values consistently indicate lower error. Negative values are expected when MSE exceeds 1 and do not denote a negative error. Because MSE depends on the scale of each molecular readout, its absolute transformed value should be compared among methods within a dataset, not across datasets. Reported summary values are obtained by averaging these per-perturbation scores over all held-out perturbations and then across folds. Because these global metrics are dominated by highly abundant and hub features that vary little across perturbations, they can overstate predictive accuracy for perturbation-specific effects. We therefore also compute $\mathrm{PCC}_{\mathrm{DE}}$, $\mathrm{MSE}_{\mathrm{DE}}$, and R^2_{DE} over the observed differentially expressed genes for each perturbation.

DEG recovery metrics.

For each held-out perturbation and molecular readout, we rank observed genes using `scanpy.tl.rank_genes_groups`. Genes are ordered by the absolute test statistic, with `rankby_abs=True` and `use_raw=False`. The established scRNA-seq benchmarks use Scanpy's `t-test_overestim_var` option. The under-studied multimodal benchmarks use the Wilcoxon option. The top 50 observed differentially expressed genes define the positive set. For a readout with p shared genes, its size is $\min(50, \lfloor p/2 \rfloor, p-1)$. These sets are fixed within each evaluation fold and shared across all methods and CellForge runs. They are used only for post hoc evaluation.

For every method, predicted genes are ranked by the absolute difference between the predicted perturbation mean and the observed control mean. Recovery@50 is labelled "Recall" in the ranking heatmaps. It is the overlap between the top 50 observed and predicted differentially expressed genes, divided by the number of genes compared. ROC curves use membership in the observed differentially expressed gene set as the binary outcome. The absolute predicted change is the continuous score across all shared genes, and AUROC summarizes the curves.

Human involvement.

Human review is intentionally minimal and limited to two steps. The first occurs after Ideation: a reviewer confirms that the research plan is scientifically plausible and internally coherent, covering dataset parsing, split definition, and benchmark feasibility, without modifying the plan content. The second occurs before any GPU job is launched: the reviewer checks that the training script matches the approved plan and complies with local HPC usage policy, to prevent harmful or resource-wasteful execution. Failed checks at either step trigger regeneration rather than manual editing.

Blinded planning evaluation.

To separate plan quality from downstream code execution, we evaluated anonymized CellForge and baseline research plans using two blinded review procedures. A research plan is the Ideation-stage output and specifies task analysis, preprocessing, neural architecture, loss function, and training protocol. First, five independent LLM judges (Claude 3.7, DeepSeek-R1, OpenAI o1, Qwen-2.5, and Llama 3.1) scored each plan from 1 to 10 for scientific validity, technical feasibility, innovation, experimental design, and impact potential. We removed formatting cues to reduce style bias and randomized plan order in each evaluation round. Scores were averaged across five independent rounds, and pairwise Pearson correlations were used to quantify inter-judge reliability. Second, three domain experts in single-cell biology performed a separate blinded review of the same anonymized plans, scoring technical novelty, feasibility, clarity and consistency, biological plausibility, mechanism explanation, pathway relevance, and cross-perturbation generalizability. We used Pearson correlation to compare mean human-expert scores with the corresponding aggregate LLM-judge scores and, for CellForge proposals, with the graph-based discussion confidence score. For phase-resolved visualization,

task-analysis scores summarized dataset analysis, task-type analysis, and baseline-defect analysis, whereas method-design scores summarized scientific validity, technical feasibility, innovation, experimental design, and impact potential. The human review did not alter the plans or participate in CellForge's design loop.

Formulation

The Formulation stage converts the processed dataset, task description, and community-agreed evaluation criterion into a well-defined research question and standardized benchmark setting that grounds all subsequent design and evaluation decisions.

Data parsing and benchmark definition.

The system reads processed expression matrices (cell-by-feature count tables) across modalities (scRNA-seq, scATAC-seq, CITE-seq). The upstream step of converting raw sequencing reads into count matrices has already been automated by agentic pipelines [69] and could be composed with CellForge to span the full path from FASTQ to trained model; we do not pursue this here, as all benchmarks used in this study provide expression-level data. From these matrices, the system identifies perturbation, feature, and context columns, extracts the summary statistics needed to define a valid prediction task, and constructs a unified five-fold split over perturbation identities. It also checks that each fold preserves sufficient cellular coverage for training and testing. This component outputs a standardized dataset report and split definition; a representative parsed summary and the associated prompt structure are provided in Supplementary Sections C.1.

Baseline benchmarking.

The Formulation stage consults the seed papers supplied as an input to identify baseline families relevant to the task specification, reproduces those baselines under the same five-fold split, and records a reference benchmark before any new architecture is designed. Because baseline comparisons are sensitive to implementation details, a human reviewer verifies that the selected baselines, code paths, and evaluation setup are appropriate for the task before these reference results are used downstream. This stage outputs the task report, split definition, and baseline performance context used by the Ideation stage.

Ideation

The Ideation stage translates the task report, retrieval results, and baseline evidence into a concrete, literature-grounded modeling plan through structured expert discussion. The goal is to identify a dataset-matched design space, not to tune hyperparameters of a fixed template.

Literature retrieval.

At the start of research ideation, the retrieval system combines the seed corpus of key papers provided as an input (46 papers across the benchmarks reported here) with dynamic PubMed and web searches that extend the seed set. The local corpus is indexed in Qdrant for embedding-based nearest-neighbor lookup and provides stable coverage of benchmark papers, perturbation-response models, multimodal methods, foundation models, and implementation resources. Candidate documents are ranked by cosine similarity over Sentence-BERT embeddings [70], and the search alternates between breadth-first exploration and depth-first follow-up so that the retrieved set covers both diverse modeling families and practical implementation details. We use this component as a retrieval heuristic rather than evaluate it as a standalone retrieval benchmark. In practice, the retrieval stage applies duplicate pruning and minimum relevance thresholds to control search breadth while retaining implementation-specific evidence. The full fixed-corpus inventory is provided in Supplementary Section B.

Critic-mediated expert panel.

Within any single run, all experts share the same LLM backend and differ only in their role-specific system prompts, so that design variation reflects the multi-agent protocol rather than differences between foundation models. Across experiments, we evaluated five LLM backends under matched decoding parameters. For a

given task, the system selects a subset of experts such as a data-modeling expert, a single-cell biology expert, and a deep-learning expert, together with a critic that reads all proposals and provides structured feedback. Operationally, experts do not write code in this stage; they iteratively refine model plans, including encoder structure, perturbation representation, loss design, and training strategy. For each expert's current proposal, the panel tracks a coordination score $c_t^{(i)} \in [0, 1]$, where i indexes the experts and t the discussion round; it guides deliberation and stopping but is neither a calibrated probability nor an experimental performance measure. The protocol mirrors conference review. Each expert first acts as an author by submitting a plan, then as a reviewer by assessing the other plans. The central critic acts as an area chair, synthesizing peer reviews into a meta-review and score for each plan. Experts revise their proposals in the next round, and only the selected final plan advances to Execution.

Discussion protocol.

The discussion implements this conference-review process over at most $T_{\max} = 20$ synchronous rounds. Each round has four steps. First, every expert acts as an author and submits a revised plan $m_t^{(i)}$, conditioned on the task report, retrieved evidence, and earlier feedback. Second, every other expert j (where $j \neq i$) acts as a reviewer, returning written comments and a peer score $r_{\text{peer},t}^{(i,j)} \in [0.1, 1.0]$. Third, the central critic acts as an editor or area chair. It reads the proposals and peer reviews, evaluates scientific merit, technical feasibility, biological validity, comparison with published methods, and implementation strategy, and issues a meta-review with an overall score $r_{\text{crit},t}^{(i)} \in [0.1, 1.0]$. Fourth, the authors receive the peer reviews and meta-review before revising their plans for the next round. These scores are LLM judgments produced by role-specific review prompts, not measurements on held-out data. The panel combines them in a fixed weighted moving average,

$$c_t^{(i)} = 0.3\, c_{t-1}^{(i)} + 0.4\, r_{\text{crit},t}^{(i)} + \frac{0.3}{k-1} \sum_{j \neq i} r_{\text{peer},t}^{(i,j)}. \tag{1}$$

Here $c_0^{(i)}$ is the initial score before review, which is 0.5, and k is the number of experts. The update assigns 40% weight to the critic's editorial assessment, 30% to the mean peer assessment, and 30% to the previous-round score. It ranks proposals by panel deliberation rather than held-out predictive performance. After each round, the coordination scores identify weaker proposals and guide the critic's next meta-review and the authors' revisions. All candidate plans remain under discussion until every proposal has $c_t^{(i)} \geq 0.8$ and the largest pairwise score difference is below 0.03. If this convergence criterion is not met by $T_{\max}$, the process stops at the round limit. At termination, the critic uses the final ranking and accumulated reviews to make an editor-like decision. Lower-ranked alternatives are excluded, and only the selected final plan is incorporated into the modeling brief and advanced to Execution.

The Ideation stage outputs the task report, split definition, baseline performance context, and an approved research plan (preprocessing strategy, neural architecture, loss function, and training protocol) that together define a dataset-matched design space for the Execution stage.

Execution

The Execution stage converts the approved research plan into runnable code, submits approved training jobs, and validates results. A detailed compute profile and cost estimation, including token counts, GPU hours, and a provider-agnostic price-conversion formula, is provided in Supplementary Section A.3.

Code generation and repair.

We implement code generation with OpenHands [46], a software-engineering agent that iteratively proposes actions, executes them in a sandboxed workspace, observes the resulting outputs, and revises subsequent actions. Given the approved research plan, the Code Generator writes Python training code with pinned dependencies, executes the code, inspects runtime feedback, applies patches, and re-executes. Code generation follows a ReAct-style loop: propose action, execute, observe output, revise [71]. This loop repeats for up to 5 attempts. If all attempts fail, the system rolls back to the last passing state; runs that fail to produce a

trainable model after exhausting all repair attempts are excluded from the reported results and replaced by a new independent run, so that all summary statistics reflect three successfully completed design-train cycles. A breakdown of resolved error categories is provided in Supplementary Section A.2.

Training and validation loop.

After approval, the scheduler launches training with 5-fold cross-validation, early stopping (patience = 10 epochs), a cosine-annealing learning-rate schedule, and periodic checkpointing. Within each fold, early stopping monitors validation loss every epoch and halts training if no improvement is observed for 10 consecutive epochs. After each fold, the Validation Agent evaluates the best checkpoint with the perturbation-level metric pipeline defined above and summarizes failure modes such as systematic under-prediction of large-magnitude responses or instability on specific perturbation families. Across folds, the Validation Agent applies a second-level check: if validation loss increases for 3 consecutive folds, it adjusts regularization strength or reduces the number of training epochs and restarts from the best checkpoint. These validation summaries are then fed back to the Code Generator for a subsequent repair round.

Data availability

All datasets used in this study are publicly available. The Adamson et al. [26], Norman et al. [27], and Srivatsan et al. [28] datasets are available at NCBI GEO under accessions GSE90546, GSE133344, and GSE139944, respectively. Pre-processed `.h5ad` files for these datasets are also available through the scPerturb resource [30] under DOI `10.5281/zenodo.13350497`.

Code availability

The source code of CellForge is publicly available at `github.com/gersteinlab/CellForge`. All reported model training runs were executed through Slurm on Yale Center for Research Computing infrastructure, primarily on the `misha` and `bouchet` clusters, with production jobs submitted via `sbatch`.

Funding

This work was supported by the ALW Professorship Funds.

Author contributions

X.T., J.C., Y.S., and Z.Y. contributed equally to this work. X.T., J.C., Z.Y., Y.S., and Y.C. conceived the study. X.T. led the project. J.C., Z.Y., and Y.S. developed the CellForge multi-agent framework. J.C. implemented and benchmarked all single-cell baseline methods. J.C., Z.Y., Y.S., and Y.C. analyzed and interpreted model outputs. Y.S., J.C., and Z.Y. created the figures. W.W. contributed to experimental design and downstream analysis directions. F.W., Y.Zh., W.S., Z.H., A.C., X.L., S.K., and F.T. reviewed the manuscript and provided feedback. X.T. and M.G. supervised the project. Y.S., J.C., X.T., and Z.Y. wrote the manuscript. All authors read and approved the final manuscript.

Acknowledgements

The authors thank the Yale Center for Research Computing for providing computational resources and technical support.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary Information is available for this paper. Correspondence and requests for materials should be addressed to X.T. or M.G.

References

[1] Bunne, C. *et al.* How to build the virtual cell with artificial intelligence: Priorities and opportunities. *Cell* **187**, 7045–7063 (2024). URL https://doi.org/10.1016/j.cell.2024.11.015.

[2] Roohani, Y. H. *et al.* Virtual cell challenge: Toward a turing test for the virtual cell. *Cell* **188**, 3370–3374 (2025). URL https://doi.org/10.1016/j.cell.2025.04.027.

[3] Cui, H. *et al.* Towards multimodal foundation models in molecular cell biology. *Nature* **640**, 623–633 (2025). URL https://doi.org/10.1038/s41586-025-08710-y.

[4] Lotfollahi, M., Wolf, F. A. & Theis, F. J. scGen predicts single-cell perturbation responses. *Nature Methods* **16**, 715–721 (2019). URL https://doi.org/10.1038/s41592-019-0494-8.

[5] Lotfollahi, M. *et al.* Predicting cellular responses to complex perturbations in high-throughput screens. *Molecular Systems Biology* **19**, e11517 (2023). URL https://doi.org/10.15252/msb.202211517.

[6] Piran, Z., Cohen, N., Hoshen, Y. & Nitzan, M. Disentanglement of single-cell data with biolord. *Nature Biotechnology* **42**, 1678–1683 (2024). URL https://doi.org/10.1038/s41587-023-02079-x.

[7] Lotfollahi, M. *et al.* Predicting cellular responses to complex perturbations in high-throughput screens. *Molecular Systems Biology* **19**, e11517 (2023). URL https://doi.org/10.15252/msb.202211517.

[8] Bunne, C., Krause, A. & Cuturi, M. Supervised training of conditional monge maps. *Advances in Neural Information Processing Systems* **35**, 6859–6872 (2022). URL https://doi.org/10.48550/arXiv.2206.14262.

[9] Bunne, C. *et al.* Learning single-cell perturbation responses using neural optimal transport. *Nature Methods* **20**, 1759–1768 (2023). URL https://doi.org/10.1038/s41592-023-01987-9.

[10] Roohani, Y., Huang, K. & Leskovec, J. Predicting transcriptional outcomes of novel multigene perturbations with GEARS. *Nature Biotechnology* **42**, 927–935 (2024). URL https://doi.org/10.1038/s41587-023-01905-6.

[11] Klein, D. *et al.* Cellflow enables generative single-cell phenotype modeling with flow matching. *bioRxiv* 2025–04 (2025). URL https://doi.org/10.1101/2025.04.11.648220.

[12] Cui, H. *et al.* scGPT: toward building a foundation model for single-cell multi-omics using generative ai. *Nature Methods* **21**, 1470–1480 (2024). URL https://doi.org/10.1038/s41592-024-02201-0.

[13] Adduri, A. K. *et al.* Predicting cellular responses to perturbation across diverse contexts with state. *bioRxiv* (2025). URL https://doi.org/10.1101/2025.06.26.661135.

[14] Luecken, M. D. *et al.* Defining and benchmarking open problems in single-cell analysis. *Nature Methods* **19**, 412–420 (2022). URL https://doi.org/10.1038/s41592-022-01409-2.

[15] Luecken, M. D. *et al.* Defining and benchmarking open problems in single-cell analysis. *Nature Biotechnology* **43**, 1035–1040 (2025). URL https://doi.org/10.1038/s41587-025-02694-w.

[16] Gao, Y. *et al.* Toward subtask decomposition-based learning and benchmarking for predicting genetic perturbation outcomes and beyond. *Nature Computational Science* **4**, 773–785 (2024). URL https://doi.org/10.1038/s43588-024-00698-1.

[17] Wei, Z. *et al.* Benchmarking algorithms for generalizable single-cell perturbation response prediction. *Nature Methods* **23**, 451–464 (2025). URL https://doi.org/10.1038/s41592-025-02980-0.

[18] Viñas Torné, R. *et al.* Systema: a framework for evaluating genetic perturbation response prediction beyond systematic variation. *Nature Biotechnology* (2025). URL https://doi.org/10.1038/s41587-025-02777-8.

[19] Squair, J. W. & Theis, F. J. A practical guide to single-cell perturbation analysis. *Nature Reviews Genetics* (2024). URL https://doi.org/10.1038/s41576-024-00713-x.

[20] Ahlmann-Eltze, C., Huber, W. & Anders, S. Deep-learning-based gene perturbation effect prediction does not yet outperform simple linear baselines. *Nature Methods* **22**, 1657–1661 (2025). URL https://doi.org/10.1038/s41592-025-02772-6.

[21] Liscovitch-Brauer, N. *et al.* Profiling the genetic determinants of chromatin accessibility with scalable single-cell crispr screens. *Nature Biotechnology* **39**, 1270–1277 (2021). URL https://doi.org/10.1038/s41587-021-00902-x.

[22] Papalexi, E. *et al.* Characterizing the molecular regulation of inhibitory immune checkpoints with multimodal single-cell screens. *Nature Genetics* **53**, 322–331 (2021). URL https://doi.org/10.1038/s41588-021-00778-2.

[23] Gayoso, A. *et al.* Joint probabilistic modeling of single-cell multi-omic data with totalVI. *Nature Methods* **18**, 272–282 (2021). URL https://doi.org/10.1038/s41592-020-01050-x.

[24] Ashuach, T. *et al.* MultiVI: deep generative model for the integration of multimodal data. *Nature Methods* **20**, 1222–1231 (2023). URL https://doi.org/10.1038/s41592-023-01909-9.

[25] Saunders, R. A. *et al.* Perturb-multimodal: A platform for pooled genetic screens with imaging and sequencing in intact mammalian tissue. *Cell* **188**, 4790–4809.e22 (2025). URL https://doi.org/10.1016/j.cell.2025.05.022.

[26] Adamson, B. *et al.* A multiplexed single-cell CRISPR screening platform enables systematic dissection of the unfolded protein response. *Cell* **167**, 1867–1882 (2016). URL https://doi.org/10.1016/j.cell.2016.11.048.

[27] Norman, T. M. *et al.* Exploring genetic interaction manifolds constructed from rich single-cell phenotypes. *Science* **365**, 786–793 (2019). URL https://doi.org/10.1126/science.aax4438.

[28] Srivatsan, S. R. *et al.* Massively multiplex chemical transcriptomics at single-cell resolution. *Science* **367**, 45–51 (2020). URL https://doi.org/10.1126/science.aax6234.

[29] Cui, A. *et al.* Dictionary of immune responses to cytokines at single-cell resolution. *Nature* **625**, 377–384 (2024). URL https://doi.org/10.1038/s41586-023-06816-9.

[30] Peidli, S. *et al.* scPerturb: harmonized single-cell perturbation data. *Nature Methods* **21**, 531–540 (2024). URL https://doi.org/10.1038/s41592-023-02144-y.

[31] Zhu, R. *et al.* Genome-scale perturb-seq in primary human cd4+ t cells maps context-specific regulators of t cell programs and human immune traits. *bioRxiv* 2025.12.23.696273 (2025). URL https://doi.org/10.64898/2025.12.23.696273.

[32] Zhang, J. *et al.* Tahoe-100m: A giga-scale single-cell perturbation atlas for context-dependent gene function and cellular modeling. *bioRxiv* 2025.02.20.639398 (2025). URL https://doi.org/10.1101/2025.02.20.639398.

[33] Yu, H., Qian, W., Song, Y. & Welch, J. D. PerturbNet predicts single-cell responses to unseen chemical and genetic perturbations. *Molecular Systems Biology* **21**, 960–982 (2025). URL https://doi.org/10.1038/

s44320-025-00131-3.

[34] Li, Y. *et al.* Genetic-to-chemical perturbation transfer learning through unified multimodal molecular representations. *bioRxiv* (2025). URL https://doi.org/10.1101/2025.02.02.635055.

[35] Heumos, L. *et al.* Best practices for single-cell analysis across modalities. *Nature Reviews Genetics* **24**, 550–572 (2023). URL https://doi.org/10.1038/s41576-023-00586-w.

[36] Gao, S. *et al.* Empowering biomedical discovery with AI agents. *Cell* **187**, 6125–6151 (2024). URL https://doi.org/10.1016/j.cell.2024.09.022.

[37] Swanson, K., Wu, W., Bulaong, N. L., Pak, J. E. & Zou, J. The virtual lab of AI agents designs new SARS-CoV-2 nanobodies. *Nature* **646**, 716–723 (2025). URL https://doi.org/10.1038/s41586-025-09442-9.

[38] Huang, K. *et al.* Biomni: A general-purpose biomedical ai agent. *bioRxiv* 2025–05 (2025). URL https://doi.org/10.1101/2025.05.30.656746.

[39] Si, C., Yang, D. & Hashimoto, T. Can LLMs generate novel research ideas? a large-scale human study with 100+ NLP researchers. *International Conference on Learning Representations* (2025). URL https://doi.org/10.48550/arXiv.2409.04109.

[40] Roohani, Y. H., Vora, J., Huang, Q., Liang, P. & Leskovec, J. BioDiscoveryAgent: An ai agent for designing genetic perturbation experiments (2024). URL https://doi.org/10.48550/arXiv.2405.17631. ICLR 2024 Workshop on Machine Learning for Genomics Explorations.

[41] Schmidgall, S. *et al.* Agent laboratory: Using LLM agents as research assistants. *Findings of the Association for Computational Linguistics: EMNLP 2025* 5977–6043 (2025). URL https://doi.org/10.18653/v1/2025.findings-emnlp.320.

[42] Majumder, B. P. *et al.* DiscoveryBench: Towards data-driven discovery with large language models. *International Conference on Learning Representations* (2025). URL https://doi.org/10.48550/arXiv.2407.01725.

[43] Panigrahi, S. S., Videnovi'c, J. & Brbi'c, M. HeurekaBench: A benchmarking framework for AI co-scientist. *International Conference on Learning Representations* (2026). URL https://doi.org/10.48550/arXiv.2601.01678.

[44] Yang, J. *et al.* Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* **37**, 50528–50652 (2024).

[45] Li, Y. *et al.* Competition-level code generation with alphacode. *Science* **378**, 1092–1097 (2022). URL https://doi.org/10.1126/science.abq1158.

[46] Wang, X. *et al.* OpenHands: An open platform for AI software developers as generalist agents. *International Conference on Learning Representations* (2025). URL https://doi.org/10.48550/arXiv.2407.16741.

[47] Jimenez, C. E. *et al.* SWE-bench: Can language models resolve real-world GitHub issues? *International Conference on Learning Representations* (2024). URL https://doi.org/10.48550/arXiv.2310.06770.

[48] Chan, J. S. *et al.* MLE-bench: Evaluating machine learning agents on machine learning engineering. *International Conference on Learning Representations* (2025). URL https://doi.org/10.48550/arXiv.2410.07095.

[49] Wang, Z. *et al.* GeneAgent: self-verification language agent for gene-set analysis using domain databases. *Nature Methods* **22**, 1677–1685 (2025). URL https://doi.org/10.1038/s41592-025-02748-6.

[50] Qu, Y. *et al.* CRISPR-GPT for agentic automation of gene-editing experiments. *Nature Biomedical Engineering* **10**, 245–258 (2026). URL https://doi.org/10.1038/s41551-025-01463-z.

[51] Alber, S. *et al.* CellVoyager: AI CompBio agent generates new insights by autonomously analyzing biological data. *Nature Methods* (2026). URL https://doi.org/10.1038/s41592-026-03029-6.

[52] Penad'es, J. R. *et al.* Ai mirrors experimental science to uncover a mechanism of gene transfer crucial to bacterial evolution. *Cell* **188**, 6654–6665.e2 (2025). URL https://doi.org/10.1016/j.cell.2025.08.018.

[53] Lu, C. *et al.* Towards end-to-end automation of AI research. *Nature* **651**, 914–919 (2026). URL https://doi.org/10.1038/s41586-026-10265-5.

[54] Tang, L. Artificial intelligence agents for biology. *Nature Methods* **22**, 2496–2497 (2025). URL https://doi.org/10.1038/s41592-025-02958-y.

[55] Lotfollahi, M., Wolf, F. A. & Theis, F. J. scGen predicts single-cell perturbation responses. *Nature Methods* **16**, 715–721 (2019). URL https://doi.org/10.1038/s41592-019-0494-8.

[56] Cui, H. *et al.* scGPT: Toward building a foundation model for single-cell multi-omics using generative AI. *Nature Methods* **21**, 1470–1480 (2024). URL https://doi.org/10.1038/s41592-024-02201-0.

[57] Hetzel, L. *et al.* Predicting cellular responses to novel drug perturbations at a single-cell resolution. *Advances in Neural Information Processing Systems* **35**, 26711–26722 (2022). URL https://doi.org/10.48550/arXiv.2204.13545.

[58] Roohani, Y., Huang, K. & Leskovec, J. Predicting transcriptional outcomes of novel multigene perturbations with GEARS. *Nature Biotechnology* **42**, 927–935 (2024). URL https://doi.org/10.1038/s41587-023-01905-6.

[59] Bai, D., Ellington, C. N., Mo, S., Song, L. & Xing, E. P. AttentionPert: accurately modeling multiplexed genetic perturbations with multi-scale effects. *Bioinformatics* **40**, i453–i461 (2024). URL https://doi.org/10.1093/bioinformatics/btae244.

[60] Schiebinger, G. *et al.* Optimal-transport analysis of single-cell gene expression identifies developmental trajectories in reprogramming. *Cell* **176**, 928–943 (2019). URL https://doi.org/10.1016/j.cell.2019.01.006.

[61] Ho, J., Jain, A. & Abbeel, P. Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems* **33**, 6840–6851 (2020).

[62] OpenAI. Introducing deep research. OpenAI product announcement (2025). URL https://openai.com/index/deep-research/. Accessed: 2025-05-08.

[63] AI, P. Deep research. Perplexity AI product announcement (2025). URL https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research. Accessed: 2025-05-08.

[64] DeepMind, G. Gemini deep research. Google product announcement (2025). URL https://gemini.google/overview/deep-research/. Accessed: 2025-05-08.

[65] Hao, Q., Xu, F., Li, Y. & Evans, J. Artificial intelligence tools expand scientists' impact but contract science's focus. *Nature* **649**, 1237–1243 (2026). URL https://doi.org/10.1038/s41586-025-09922-y.

[66] Binan, L. *et al.* Simultaneous CRISPR screening and spatial transcriptomics reveal intracellular, intercellular, and functional transcriptional circuits. *Cell* **188**, 2141–2158.e18 (2025). URL https://doi.org/10.1016/j.cell.2025.02.012.

[67] Shen, K. *et al.* Spatial perturb-seq: single-cell functional genomics within intact tissue architecture. *Nature Communications* (2026). URL https://doi.org/10.1038/s41467-026-69677-6.

[68] Arc Institute. cell-eval: Comprehensive suite for evaluating perturbation prediction models. https://github.com/ArcInstitute/cell-eval (2026). GitHub repository, accessed March 19, 2026.

[69] Youngblut, N. D. *et al.* scBaseCamp: An AI agent-curated, uniformly processed, and continually expanding single cell data repository (2025). URL https://doi.org/10.1101/2025.02.27.640494.

[70] Reimers, N. & Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* 3982–3992 (2019). URL https://doi.org/10.18653/v1/D19-1410.

[71] Yao, S. *et al.* ReAct: Synergizing reasoning and acting in language models. *International Conference on Learning Representations* (2023). URL https://doi.org/10.48550/arXiv.2210.03629.

[72] Veličković, P. *et al.* Graph attention networks. *International Conference on Learning Representations* (2018). URL https://openreview.net/forum?id=rJXMpikCZ.

Figure legends

Fig. 1 | CellForge as an autonomous virtual-cell model designer. a, Perturbation-prediction task. Given a control cell state X_{ctrl} and a perturbation drawn from perturbation space $\mathcal{P}$, the model predicts the corresponding post-perturbation state X_{pert}. **b**, CellForge inputs and outputs. User-specified inputs include a preprocessed single-cell perturbation dataset, a perturbation task description, an evaluation criterion, and selected seed papers. CellForge operates as a team of autonomous agents and returns model weights, an architecture specification, and source code. The generated architecture is treated as the design hypothesis because CellForge produces the architecture, loss, and training procedure end-to-end rather than receiving them from human designers. **c**, Perturbation types and molecular readouts considered in this study. The benchmark covers drug, gene-editing, and cytokine perturbations, with readouts from scRNA-seq, scATAC-seq, and paired RNA–protein CITE-seq. **d**, Conceptual view of model architecture as a hypothesis. Existing architectures and newly generated modeling hypotheses occupy a task-dependent design space, within which CellForge identifies a candidate architecture for implementation and empirical evaluation.

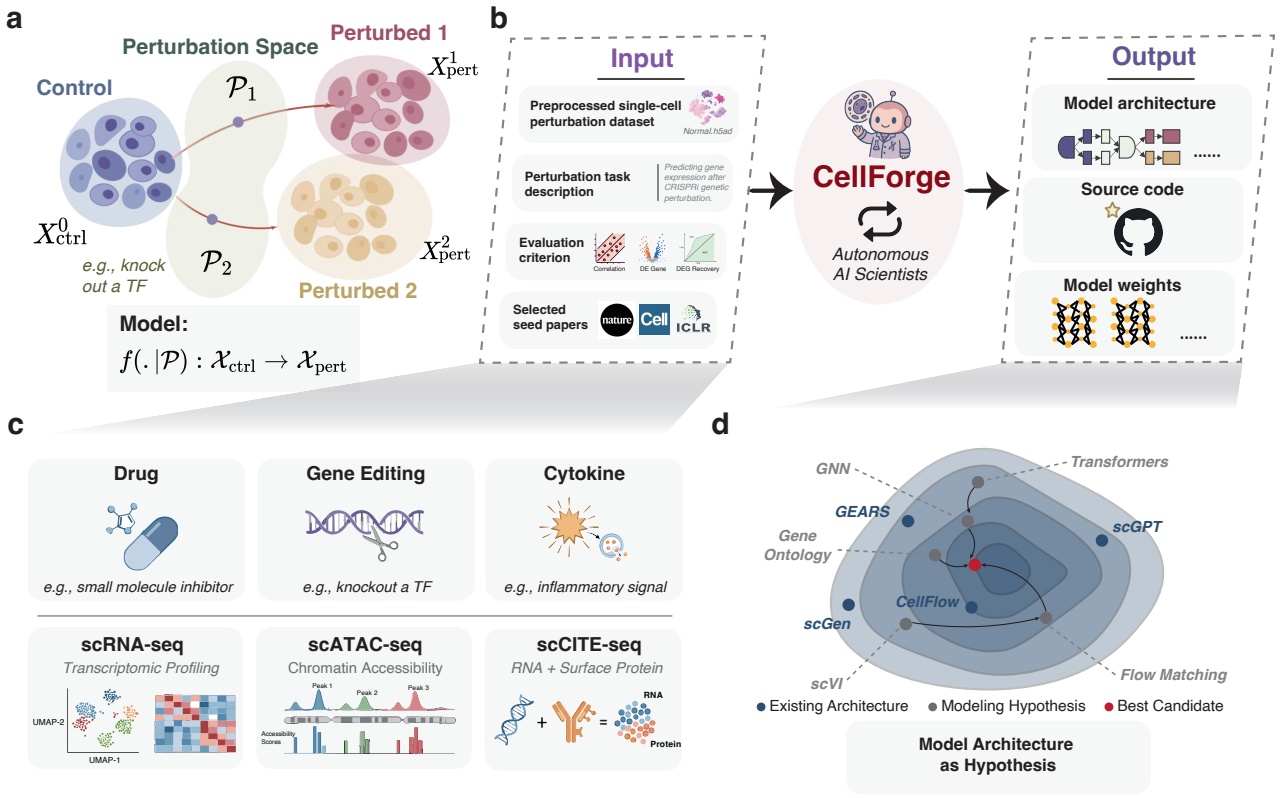
Fig. 2 | CellForge workflow for formulation, ideation, and execution. a, Task-to-hypothesis workflow. Given a biological modeling task, CellForge develops a corresponding modeling hypothesis through three stages. *Formulation* performs dataset analysis, task overview, baseline reproduction, and benchmark definition to produce a task-analysis report. *Ideation* retrieves literature and coordinates iterative expert discussion to produce a research plan. *Execution* generates code, trains models, and iteratively refines implementations to produce the final model, code, and results. **b**, Research-plan discussion and code implementation. Data-modeling, single-cell biology, and deep-learning experts propose candidate plans. A critic evaluates the proposals, and confidence-guided peer discussion and revision produce a final plan specifying preprocessing, model architecture, and loss. Coding agents then implement and execute the plan, iteratively refine candidate implementations, and return executable code, model checkpoints, and predicted responses. **c**, Confidence-score trajectories during ideation for the Norman, Adamson, and Srivatsan benchmarks. Scores are shown by discussion round for the data-modeling expert, single-cell biology expert, deep-learning expert, model-training expert, and critic agent, with divergence, convergence, and stable phases indicated. Each score aggregates the previous-round score, critic assessment, and peer assessment; it is neither a calibrated probability nor a predictive-performance measure.

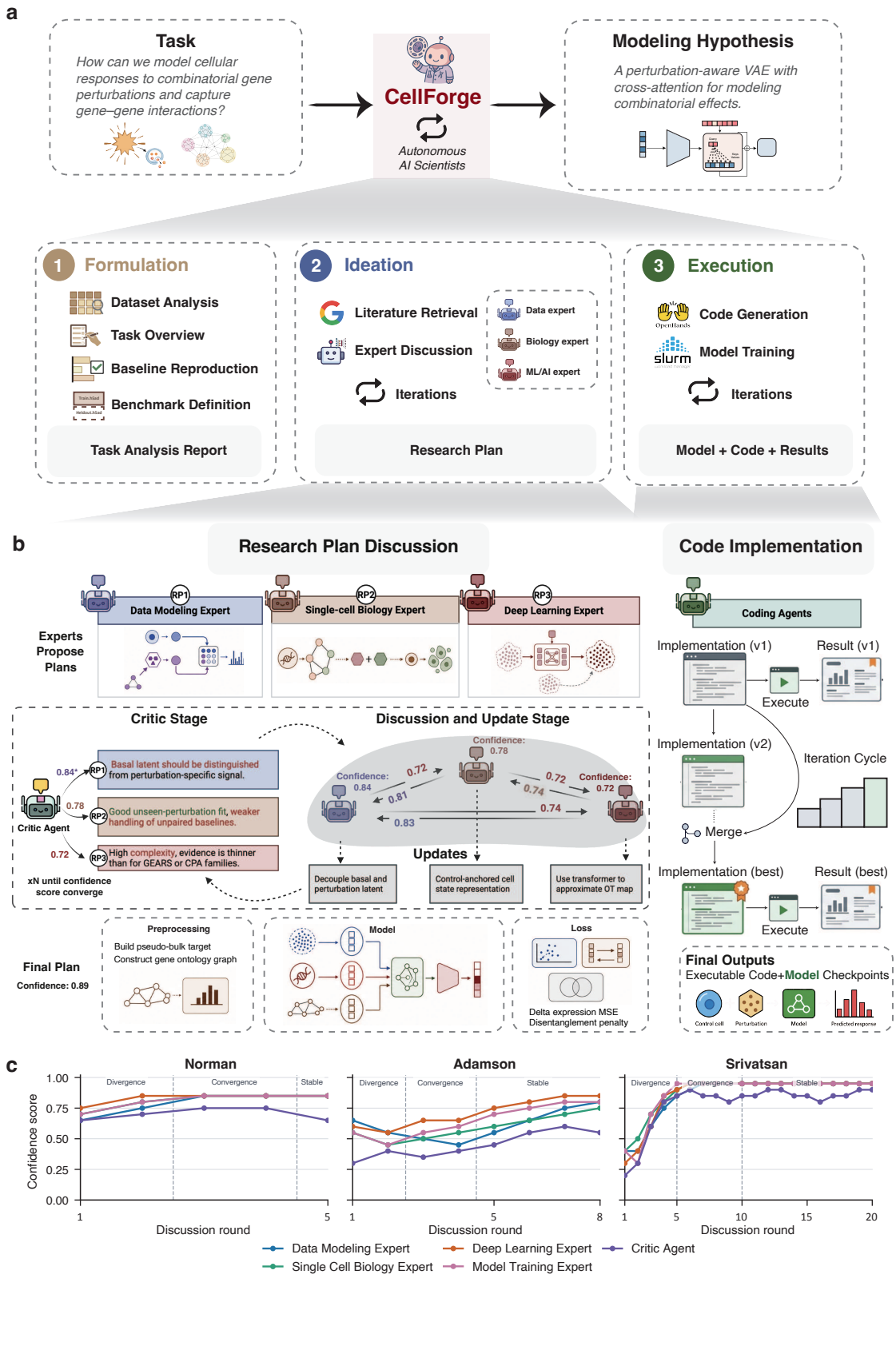
Fig. 3 | CellForge architectures benchmarked on established scRNA-seq perturbation-prediction tasks. a, *CPA-X* architecture for single-gene CRISPRi, in which an MLP expression encoder generates context embeddings and a perturbation embedding queries the encoded control-cell state by cross-attention before residual MLP decoding. **b**, *scGen-X* architecture for combinatorial CRISPRa, in which a bottlenecked expression encoder is fused with combinatorial perturbation embeddings by cross-attention before predicting the perturbed profile $\hat{x}^{\text{pert}}$. **c**, *ChemCellFlow* architecture for drug-response prediction, in which gene and chemical encoders define cell and perturbation states that are transformed by conditional optimal transport and normalizing-flow refinement before decoding $\hat{x}^{\text{pert}}$. **d**, Global and differential-expression prediction performance on the Adamson CRISPRi, Norman CRISPRa, and Srivatsan drug-response benchmarks. The upper row reports PCC and $-\log_{10}\text{MSE}$ across all genes, whereas the lower row reports PCC_{DE} and $-\log_{10}\text{MSE}_{\text{DE}}$ across observed differentially expressed genes. CellForge results are means $\pm$ s.d. across three independent runs. **e**, ROC curves for recovery of the observed differentially expressed genes. **f**, AUROC values for the recovery analysis in e. **g**, Method rankings across global prediction, differential-expression prediction and DEG-recovery metrics; lower ranks indicate better performance.

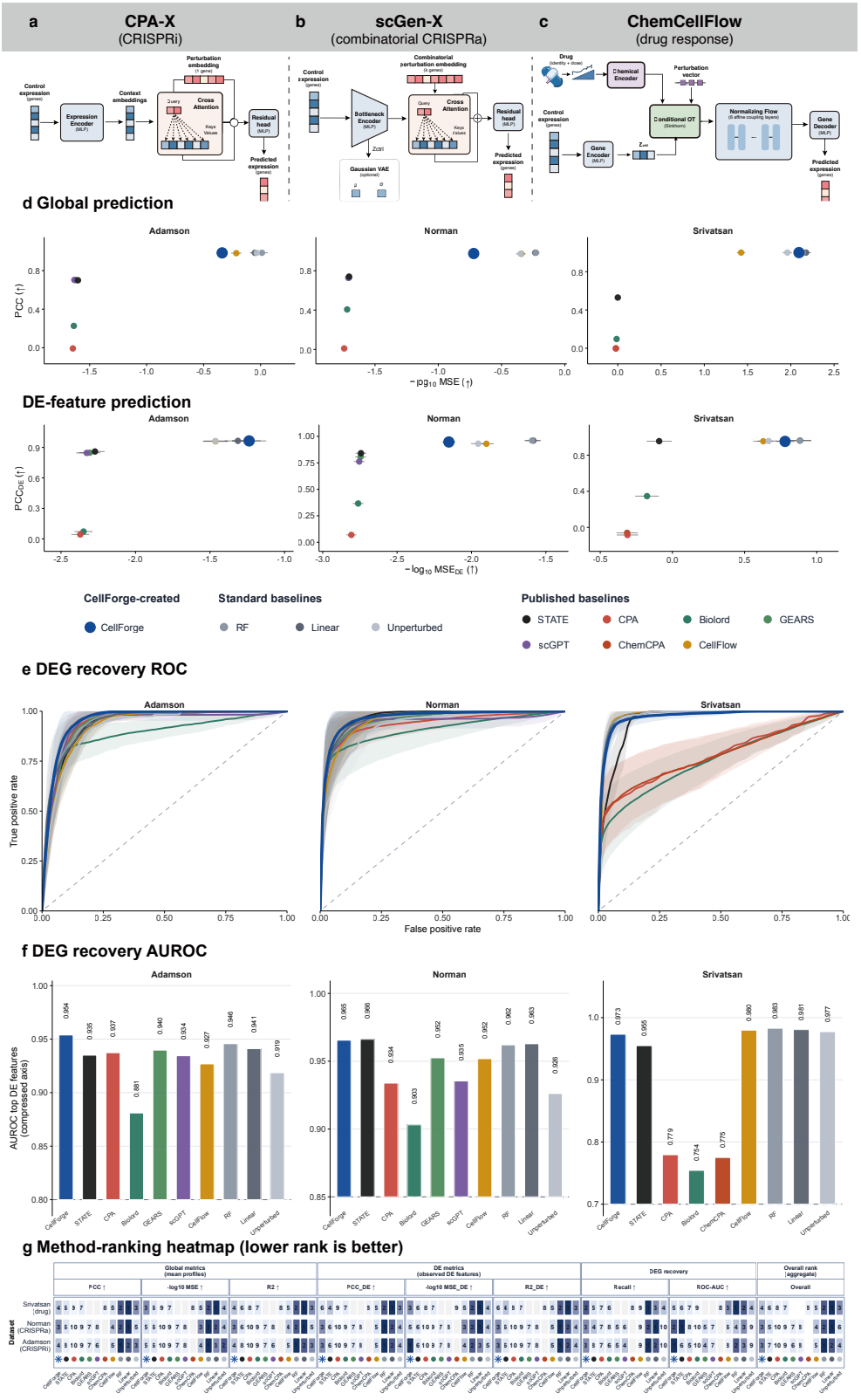
Fig. 4 | CellForge models for perturbation prediction across modalities. a, *totalGAT* architecture for paired RNA–protein CITE-seq data. Modality-specific encoders and cross-modal graph-attention layers integrate RNA features and protein markers, and separate decoders predict the perturbed RNA and protein profiles. **b**, *ChromDDPM* architecture for scATAC-seq data, using conditioning signals from cell state and perturbation context to guide denoising-based reconstruction of chromatin-accessibility profiles. **c**, *CPA-Traj* architecture for cytokine time-course data, in which a trajectory-aware variational autoencoder represents shared and

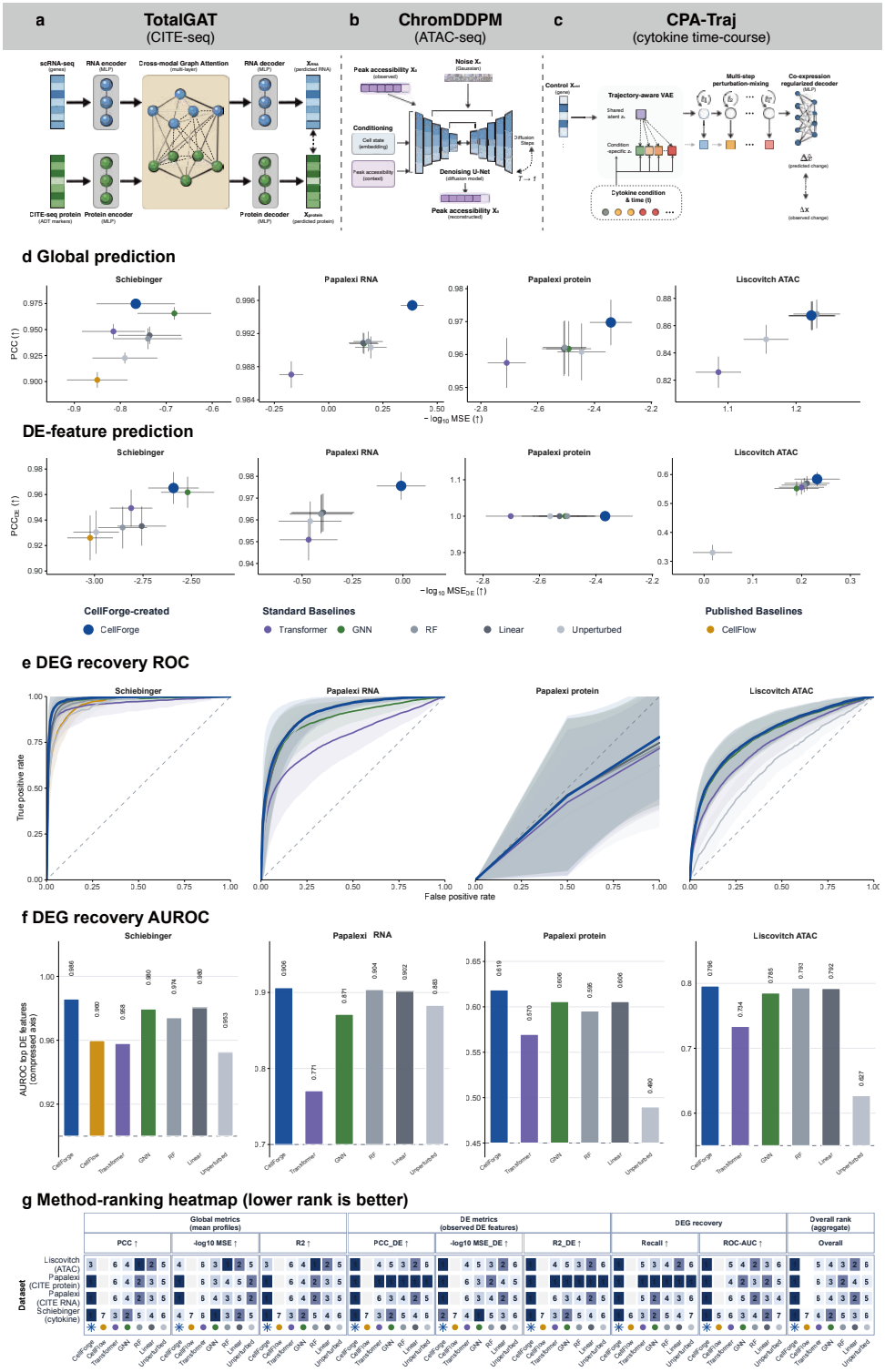
cytokine-specific cell states that are combined with time and perturbation information before predicting expression changes. **d**, Global and differential-feature prediction performance for Schiebinger RNA, Papalexi CITE-seq RNA and protein, and Liscovitch-Brauer ATAC readouts. The upper row shows PCC against $-\log_{10}\mathrm{MSE}$, and the lower row shows $\mathrm{PCC}_{\mathrm{DE}}$ against $-\log_{10}\mathrm{MSE}_{\mathrm{DE}}$. CellForge results are means $\pm$ s.d. across three independent runs. **e**, ROC curves for recovery of the observed differential features. **f**, AUROC values for the recovery analysis in e. **g**, Method rankings across global prediction, differential-feature prediction and differential-feature recovery; lower ranks indicate better performance.

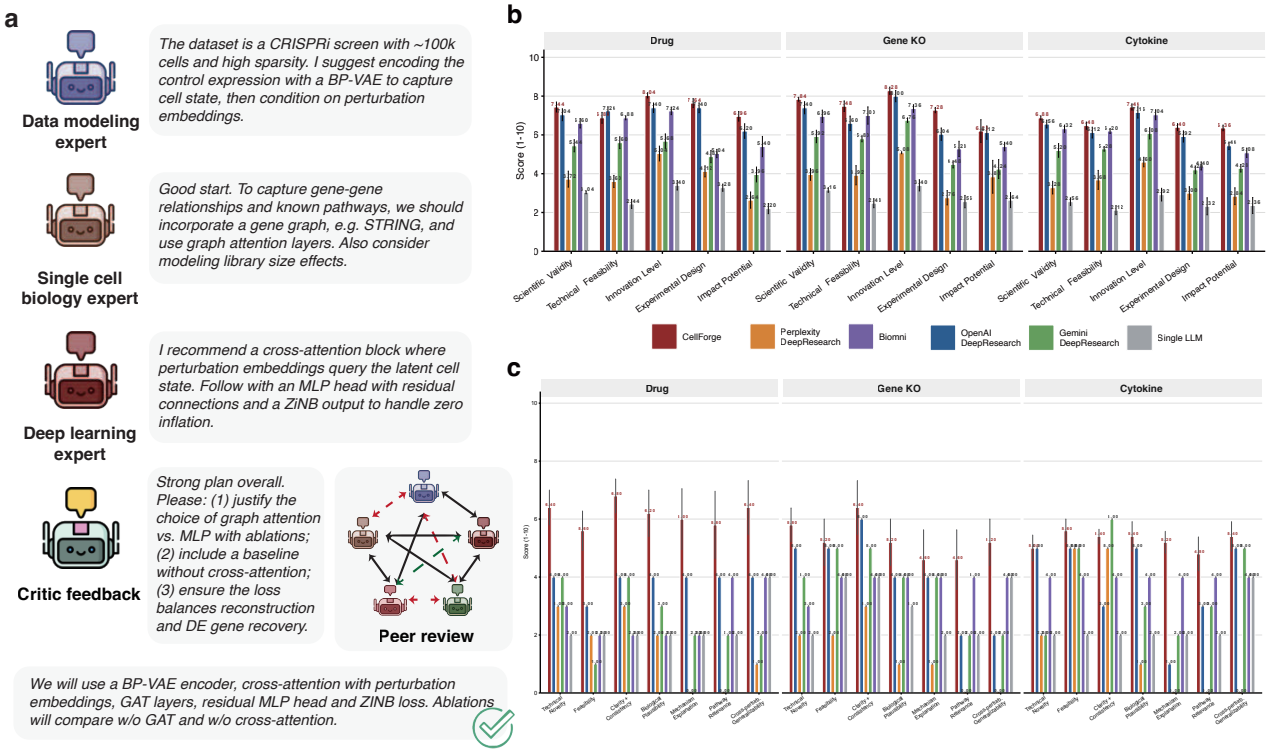
Fig. 5 | Blinded evaluation of CellForge-generated research plans. a, Example CellForge planning exchange for a single-cell perturbation task. Data-modeling, single-cell biology, and deep-learning experts propose complementary components. Peer review and critic feedback then guide the selected expert in revising the integrated plan before execution. **b**, Blinded evaluation of plans generated by CellForge, OpenAI Deep Research, Perplexity Deep Research, Gemini Deep Research, Biomni, and a single-LLM baseline across drug, gene-knockout, and cytokine tasks. Five LLM judges scored each plan from 1 to 10 for scientific validity, technical feasibility, innovation, experimental design, and impact potential. Bars show mean scores, and error bars show variability across repeated evaluations. **c**, Separate blinded human-expert evaluation of the same plans across technical novelty, feasibility, clarity and consistency, biological plausibility, mechanism explanation, pathway relevance, and cross-perturbation generalizability. Bars show mean scores across the three single-cell biology experts. **d**, Inter-judge reliability among the five LLM judges, shown as pairwise Pearson correlations between scores. Off-diagonal correlations range from 0.88 to 0.93. **e**, Associations of mean human-expert scores with CellForge's graph-based discussion confidence score for individual expert proposals (left) and with aggregate LLM-judge scores across plan-generating systems (right). Shapes denote task, and colors denote CellForge expert role (left) or plan-generating system (right). **f**, Phase-resolved evaluation profiles for gene-knockout (left), drug-treatment (middle), and cytokine (right) tasks. In each radial plot, the upper sector reports method-design criteria and the lower sector reports task-analysis criteria. Lines denote plan-generating systems, and markers denote individual LLM judges.


a
Perturbation Space
Perturbed 1
Control
$\mathcal{P}_1$
X^1_{pert}
X^0_{ctrl}
e.g., knock out a TF
$\mathcal{P}_2$
X^2_{pert}
Perturbed 2
Model:
$f(.|\mathcal{P}) : \mathcal{X}_{\text{ctrl}} \rightarrow \mathcal{X}_{\text{pert}}$
b
Input
Preprocessed single-cell perturbation dataset
Normal.h5ad
Perturbation task description
Predicting gene expression after CRISPRi genetic perturbation.
Evaluation criterion
Correlation
DE Gene
DEG Recovery
Selected seed papers
nature
Cell
ICLR
CellForge
Autonomous AI Scientists
Output
Model architecture
.....
Source code
Model weights
.....
c
Drug
e.g., small molecule inhibitor
Gene Editing
e.g., knockout a TF
Cytokine
e.g., inflammatory signal
scRNA-seq
Transcriptomic Profiling
UMAP-2
UMAP-1
scATAC-seq
Chromatin Accessibility
Peak 1
Peak 2
Peak 3
Accessibility Scores
scCITE-seq
RNA + Surface Protein
RNA
Protein
d
GNN
Transformers
GEARS
scGPT
Gene Ontology
CellFlow
scGen
scVI
Flow Matching
Existing Architecture
Modeling Hypothesis
Best Candidate
Model Architecture as Hypothesis


a
Task
How can we model cellular responses to combinatorial gene perturbations and capture gene–gene interactions?
CellForge
Autonomous AI Scientists
Modeling Hypothesis
A perturbation-aware VAE with cross-attention for modeling combinatorial effects.
1 Formulation
Dataset Analysis
Task Overview
Baseline Reproduction
Benchmark Definition
Task Analysis Report
2 Ideation
Literature Retrieval
Expert Discussion
Iterations
Data expert
Biology expert
ML/AI expert
Research Plan
3 Execution
Code Generation
Model Training
Iterations
Model + Code + Results
b
Research Plan Discussion
Code Implementation
Experts Propose Plans
RP1 Data Modeling Expert
RP2 Single-cell Biology Expert
RP3 Deep Learning Expert
Critic Stage
Discussion and Update Stage
Critic Agent
Basal latent should be distinguished from perturbation-specific signal.
Good unseen-perturbation fit, weaker handling of unpaired baselines.
High complexity, evidence is thinner than for GEARS or CPA families.
xN until confidence score converge
Updates
Decouple basal and perturbation latent
Control-anchored cell state representation
Use transformer to approximate OT map
Final Plan
Confidence: 0.89
Preprocessing
Build pseudo-bulk target
Construct gene ontology graph
Model
Loss
Delta expression MSE
Disentanglement penalty
Coding Agents
Implementation (v1)
Result (v1)
Execute
Implementation (v2)
Iteration Cycle
Merge
Implementation (best)
Result (best)
Execute
Final Outputs
Executable Code+Model Checkpoints
c
Norman
Adamson
Srivatsan
Confidence score
Discussion round
Divergence
Convergence
Stable
Data Modeling Expert
Deep Learning Expert
Critic Agent
Single Cell Biology Expert
Model Training Expert


a CPA-X (CRISPRi)
b scGen-X (combinatorial CRISPRa)
c ChemCellFlow (drug response)
d Global prediction
Adamson
Norman
Srivatsan
PCC (↑)
−log10 MSE (↑)
DE-feature prediction
PCC_DE (↑)
−log10 MSE_DE (↑)
CellForge-created
CellForge
Standard baselines
RF
Linear
Unperturbed
Published baselines
STATE
CPA
Biolord
GEARS
scGPT
ChemCPA
CellFlow
e DEG recovery ROC
True positive rate
False positive rate
f DEG recovery AUROC
AUROC top DE features (compressed axis)
g Method-ranking heatmap (lower rank is better)
Global metrics (mean profiles)
DE metrics (observed DE features)
DEG recovery
Overall rank (aggregate)
PCC ↑
-log10 MSE ↑
R2 ↑
PCC_DE ↑
-log10 MSE_DE ↑
R2_DE ↑
Recall ↑
ROC-AUC ↑
Overall
Dataset
Srivatsan (drug)
Norman (CRISPRa)
Adamson (CRISPRi)


a TotalGAT (CITE-seq)
b ChromDDPM (ATAC-seq)
c CPA-Traj (cytokine time-course)
d Global prediction
Schiebinger
Papalexi RNA
Papalexi protein
Liscovitch ATAC
PCC (↑)
−log10 MSE (↑)
DE-feature prediction
PCC_DE (↑)
−log10 MSE_DE (↑)
CellForge-created
CellForge
Standard Baselines
Transformer
GNN
RF
Linear
Unperturbed
Published Baselines
CellFlow
e DEG recovery ROC
True positive rate
False positive rate
f DEG recovery AUROC
AUROC top DE features (compressed a bit)
g Method-ranking heatmap (lower rank is better)
Global metrics (mean profiles)
DE metrics (observed DE features)
DEG recovery
Overall rank (aggregate)
PCC ↑
-log10 MSE ↑
R2 ↑
PCC_DE ↑
-log10 MSE_DE ↑
R2_DE ↑
Recall ↑
ROC-AUC ↑
Overall
Dataset
Liscovitch (ATAC)
Papalexi (CITE protein)
Papalexi (CITE RNA)
Schiebinger (cytokine)


a
Data modeling expert
The dataset is a CRISPRi screen with ~100k cells and high sparsity. I suggest encoding the control expression with a BP-VAE to capture cell state, then condition on perturbation embeddings.
Single cell biology expert
Good start. To capture gene-gene relationships and known pathways, we should incorporate a gene graph, e.g. STRING, and use graph attention layers. Also consider modeling library size effects.
Deep learning expert
I recommend a cross-attention block where perturbation embeddings query the latent cell state. Follow with an MLP head with residual connections and a ZiNB output to handle zero inflation.
Critic feedback
Strong plan overall. Please: (1) justify the choice of graph attention vs. MLP with ablations; (2) include a baseline without cross-attention; (3) ensure the loss balances reconstruction and DE gene recovery.
Peer review
We will use a BP-VAE encoder, cross-attention with perturbation embeddings, GAT layers, residual MLP head and ZINB loss. Ablations will compare w/o GAT and w/o cross-attention.
b
Drug
Gene KO
Cytokine
Score (1-10)
Scientific Validity
Technical Feasibility
Innovation Level
Experimental Design
Impact Potential
CellForge
Perplexity DeepResearch
Biomni
OpenAI DeepResearch
Gemini DeepResearch
Single LLM
c
Drug
Gene KO
Cytokine
d
Inter LLM Judges Pearson Corelation
Judge 1 (Claude 3.7)
Judge 2 (DeepSeek-R1)
Judge 3 (OpenAI o1)
Judge 4 (Qwen-plus)
Judge 5 (Llama 3.1)
1.00 0.92 0.88 0.93 0.90
0.92 1.00 0.90 0.92 0.90
0.88 0.90 1.00 0.89 0.91
0.93 0.92 0.89 1.00 0.89
0.90 0.90 0.91 0.89 1.00
e
Human Expert v.s. CellForge Confidence
r = 0.53
Human expert score
Graph-based discussion confidence score (%)
Task (shape)
Cytokine
Drug
Gene Knock-out
Agent Expert(color)
Human Expert v.s. LLM Judge's Scores
r = 0.88
LLM judge score (task analysis + method design)
Agent (color)
f
Gene Knock-out
Drug
Cytokine
Innovation Level
Technical Feasibility
Scientific Validation
Analyze Baseline Defects
Analyze Task Type
Analyze Dataset
Impact Potential
Experimental Design
CellForge
OpenAI DeepResearch
Perplexity DeepResearch
Gemini DeepResearch
Task groups
Gene Knock-out: Adamson, Norman
Drug: Srivatsan
Cytokine: Schiebinger
Axis groups
Top 5: criterion for method design
Bottom 3: criterion for task analysis
Mean score by 5 LLM judges
(Claude 3.7, DeepSeek R1, OpenAI o1, Qwen-plus, Llama 3.1) per task group × axis

A Supplementary Results

This section provides supporting analyses that complement the main results and clarifies how several claims in the main text should be interpreted.

A.1 Representative Architecture Motifs

Analysis protocol.

To examine how CellForge adapts model design to task structure, we ran the framework three times with different random seeds on each of the six benchmark datasets and inspected the executable training scripts produced by the Execution stage. The descriptions below correspond to the architectures that were actually committed to code, trained, and evaluated for each dataset. The most consistently recurring run-log label for each dataset is adopted as its canonical model name in the main text (*CPA-X*, *scGen-X*, *ChemCellFlow*, *CPA-Traj*, *totalGAT*, *ChromDDPM*); these names reflect the dominant generated configuration and should not be read as standardized or novel model classes.

Genetic Perturbation Settings

Adamson CRISPRi — CPA-X.

For single-gene CRISPRi, repeated runs converged on a feed-forward cross-attention design in which perturbation identity conditions a learned transformation of the control profile, related in spirit to scGen and CPA [5, 55]. The dominant configuration, *CPA-X*, consists of an MLP expression encoder that maps the control profile x_{ctrl} into a shared hidden space, a parallel MLP perturbation encoder for the one-hot perturbation vector, a multi-head cross-attention block in which the perturbation embedding attends to the encoded cell state, a residual MLP body, and a linear output head that predicts the post-perturbation expression profile. The model is trained with mean squared error on the perturbed profile against the observed target. Run-to-run variation was restricted in structural family: the main differences across runs were hidden width, number of attention heads, residual MLP depth, and dropout levels set by the hyperparameter search. Recurring structural motifs across runs included a single multi-head cross-attention operator between state and perturbation, a lightweight residual MLP body rather than a deep decoder stack, and absolute-profile regression rather than delta prediction.

Norman CRISPRa — scGen-X.

The combinatorial CRISPRa setting, with its paired perturbations and stronger non-additive effects, pushed generated designs toward higher-capacity perturbation–state fusion rather than simple additive shifts, closer in motivation to GEARS and AttentionPert [58, 59]. The dominant configuration, *scGen-X*, projects the high-dimensional control profile through a two-stage MLP bottleneck, optionally routes the bottleneck representation through a Gaussian VAE latent when toggled on, encodes perturbations through a linear perturbation embedding, and performs a single-token cross-attention in which the perturbation embedding queries the encoded expression state. The fused representation is refined by a residual MLP body and mapped to the predicted profile by a two-layer output head. Recurring structural motifs across runs included the expression bottleneck, the query-side perturbation cross-attention operator, and a hybrid selector that turned the VAE branch on or off depending on the run. Across seeds, combinatorial perturbation data consistently drove the design loop toward high-capacity attention-based fusion rather than simple latent shifts from a single perturbation token.

Drug Perturbation Setting

Srivatsan sci-Plex — ChemCellFlow.

The drug perturbation setting generated the widest range of proposals across seeds, consistent with the weaker and less stable benchmark performance. The recurrent motifs were nonetheless interpretable: most successful runs introduced an explicit dose-response component, an optimal-transport latent alignment step,

and a flow-based refinement stage, making the resulting designs closer in spirit to CellFlow and neural optimal transport approaches [9, 11, 57] than to generic transcriptomic autoencoders. The canonical configuration, *ChemCellFlow*, combines a gene encoder that maps the control profile to a latent z_{ctrl}; a chemical encoder that maps perturbation identity and dose through separate subencoders and fuses them into a perturbation vector p; a Sinkhorn-based conditional optimal-transport block that computes a transport plan between control and perturbed latents; a six-layer normalizing-flow refiner composed of affine coupling layers and conditioned on p; and a modular gene decoder with ten gene-group output heads and a global residual path that maps the refined latent back to gene expression. The training loss combines reconstruction, optimal transport, flow consistency, and co-expression graph regularization terms. Recurring structural motifs across runs included the separate dose-encoder branch, the OT-plus-flow latent-refinement stack, and the modular gene-group decoder.

Temporal and Multi-modal Settings

Schiebinger cytokine — CPA-Traj.

On the cytokine reprogramming dataset, the recurring pattern was not simply a larger encoder-decoder, but explicit treatment of developmental trajectory: time-aware latent structure, perturbation-conditioned mixing, and co-expression-regularized decoding. The canonical configuration, *CPA-Traj*, is built from three components. A trajectory-aware encoder embeds the control profile through an MLP and fuses it with a low-dimensional continuous time vector, then produces two parallel Gaussian latents (a shared trajectory latent and a condition-specific latent), each with its own $\mu/\log\sigma^2$ head, concatenated into a single latent z. A perturbation-mixing module embeds the one-hot perturbation linearly, concatenates it with z, and iteratively refines the mixture through three gated MLP blocks that repeatedly re-inject z as a residual. A graph-regularized decoder maps the mixed latent to the predicted expression via a three-layer MLP and applies a learned linear projection whose output is regularized by a gene–gene co-expression matrix estimated during training. Recurring structural motifs across runs included the shared-versus-condition latent split, the multi-step mixing block with z-residuals, and the co-expression regularization on the decoder output. These designs were oriented toward capturing continuous developmental dynamics rather than predicting a single end-state distribution.

Papalexi CITE-seq — totalGAT.

In the paired RNA–protein CITE-seq dataset, CellForge used *totalGAT*, a joint architecture with modality-specific encoders, perturbation fusion, graph-attention layers, and transformer-style refinement applied to multi-token projections of the fused representation. The RNA branch maps the baseline expression profile through an MLP bottleneck, concatenates it with a perturbation embedding, reshapes the fused state into learned tokens, and predicts the perturbed RNA profile with an auxiliary perturbation-reconstruction head. The protein branch encodes the baseline protein profile through residual MLP blocks, projects it into eight synthetic node tokens, passes these tokens through graph-attention layers [72], combines the pooled protein representation with a perturbation embedding through bidirectional cross-attention, and predicts the perturbed protein profile with a residual MLP head. Both readouts are trained with MSE-based objectives against the observed profile. Recurring structural motifs across runs included modality-specific encoders, multi-token projection before attention-based refinement, perturbation-aware fusion, auxiliary perturbation reconstruction, and separate RNA and protein output heads.

Chromatin Accessibility Setting

Liscovitch-Brauer scATAC-seq — ChromDDPM.

The scATAC-seq dataset produced the sharpest shift away from standard scRNA-seq modeling assumptions, with generated designs moving toward denoising diffusion rather than direct regression. The canonical configuration, *ChromDDPM*, inherits the DDPM formulation of Ho et al. [61] applied to the sparse accessibility setting: it first maps the baseline accessibility vector through a two-stage MLP bottleneck, then conditions on a sinusoidal-time embedding of the diffusion step passed through a small MLP together with a linear

Table 1: Observed failure modes in a manual review of failed CellForge executions. Percentages sum to 100%.

Failure mode	Share of reviewed failures
Computation execution error	41%
Invalid type or unsupported operation	23%
Error recovery failure	16%
Model configuration error	6%
Data access failure	5%
Other system-level failures	5%
Hallucinated structures or outputs	4%

perturbation embedding, concatenating both into a conditioning vector and projecting it to the hidden dimension. The noisy input and the conditioning vector are summed and passed through a four-block residual MLP, which produces a predicted noise vector and, through a parallel head, a predicted absolute accessibility profile; a secondary auxiliary head reconstructs the perturbation identity. The accompanying diffusion wrapper implements a standard DDPM-style forward process (linear β schedule, cumulative $\bar{\alpha}$) with q_{sample} and p_{sample} routines over up to one thousand steps. Recurring structural motifs across runs included the sinusoidal-time-plus-perturbation conditioning vector, the residual MLP denoiser body, and the auxiliary perturbation-reconstruction head used to keep the denoiser perturbation-aware. Given the modest predictive performance in this setting, we interpret *ChromDDPM* as a plausible adaptation of denoising diffusion to sparse regulatory data rather than a validated chromatin-specific architecture.

A.2 Execution failure modes

We manually reviewed 20 failed-execution cases and grouped them into recurring categories. In this small-sample review, implementation and execution issues were the most common categories observed. Table 1 summarizes this error profile and was used to guide lightweight safeguards during repair, especially explicit tensor-shape tracing and stricter validation of intermediate datatypes.

A.3 Compute profile and cost estimation

For reproducibility, we report hardware configuration and approximate token usage ranges as contextual compute information. These values are intended to convey order of magnitude rather than serve as an audited per-dataset cost analysis.

Cluster environment and scheduling.

All CellForge-designed models reported in this paper, with parameter counts between 10 million and 30 million, were trained and evaluated through Slurm on Yale Center for Research Computing infrastructure, primarily on the `misha` and `bouchet` clusters. Production runs were submitted with `sbatch`. Each job requested one GPU (`-gpus=1`) together with CPU and memory resources and used a wall-time limit of `1-00:00:00`. Iterative refinement was implemented as a sequence of Slurm jobs, so each additional repair or retraining round consumed a new one-GPU batch allocation. Because the exact node assignment varied with scheduler allocation, we report the Slurm request pattern and runtime range rather than a single hardware SKU.

Cost interpretation.

The multi-agent workflow incurs distinct LLM costs in task preparation, research ideation, and experiment execution. Table 2 reports internal development-time token-usage ranges intended to show scale rather than audited per-dataset totals. We do not report a single dollar figure because API cost depends on backend choice, service tier, and execution date. These estimates also exclude shared-cluster allocation costs, storage, and the minimal human safety checkpoints described in the main Methods. In internal runs, end-to-end execution typically completed within roughly 4–8 hours of wall-clock time once scheduled, with most of that time spent

Table 2: Illustrative token usage ranges per CellForge run, grouped by task complexity.

Phase	Simple tasks		Complex tasks	
	Prompt	Completion	Prompt	Completion
Task Preparation	15,000	50,000	25,000	100,000
Research Ideation	20,000	100,000	40,000	200,000
Experiment Execution	5,000	50,000	15,000	100,000
Total	40,000	200,000	80,000	400,000

in model training rather than agent interaction; queue delay depended on partition load and is not included in this estimate.

B Supplementary Retrieval Corpus Inventory

The following lists summarize the fixed 46-paper corpus used by Agentic Retrieval. This local paper collection is stored in Qdrant so the agents can retrieve semantically related references before deciding whether to expand outward to PubMed or the web. For readability, we group papers by functional role in the retrieval stack: benchmark and review papers, perturbation-response models, multimodal integration methods, foundation models and language-model interfaces, and resource or tooling papers. Each entry reports the paper title, a shortened author line, venue information, and a one-sentence note on why the paper is useful during retrieval and synthesis.

Benchmark and review papers.

[1] A comparison of automatic cell identification methods for single-cell RNA sequencing data. **Genome Biology (2019).**
Tamim Abdelaal, Lieke Michielsen, . . . , Marcel J. T. Reinders, Ahmed Mahfouz.
This benchmark systematically compares automated cell-type annotation methods for scRNA-seq, giving CellForge a practical reference for how downstream prediction pipelines should interpret and evaluate cell-state labels.

[2] A mini-review on perturbation modelling across single-cell omic modalities. **Computational and Structural Biotechnology Journal (2024).**
George I. Gavriilidis, Vasileios Vasileiou, . . . , Naveed Ishaque, Fotis Psomopoulos.
This review organizes the landscape of single-cell perturbation modeling across modalities, making it useful for retrieval when the agent needs a compact map of task settings, data types, and modeling choices.

[3] Benchmarking atlas-level data integration in single-cell genomics. **Nature Methods (2022).**
Malte D. Luecken, M. Büttner, . . . , M. Colomé-Tatché, Fabian J. Theis.
This paper benchmarks large-scale single-cell integration methods and evaluation metrics, which helps CellForge reason about batch correction, biological conservation, and the tradeoffs of joint embedding strategies.

[4] Benchmarking Transcriptomics Foundation Models for Perturbation Analysis: one PCA still rules them all. **arXiv (2024).**
Ihab Bendidi, Shawn Whitfield, . . . , Emmanuel Noutahi, Alisandra K. Denton.
This study shows that strong linear baselines can still outperform many transcriptomics foundation models on perturbation tasks, providing a useful caution against overcommitting retrieval to foundation-model-heavy solutions.

[5] Best Practices for Single-Cell Analysis across Modalities. **Nature Reviews Genetics (2023).**
Lukas Heumos, Anna C. Schaar, . . . , Herbert B. Schiller, Fabian J. Theis.
This best-practices review summarizes modality-aware preprocessing, quality control, integration, and interpretation decisions, so it serves as a methodological anchor for robust pipeline construction.

[6] Deep Learning Tackles Single-Cell Analysis-a Survey of Deep Learning for scRNA-seq Analysis. **Briefings in Bioinformatics (2022).**
Mario Flores, Zhentao Liu, . . . , Yidong Chen, Yufei Huang.
This survey distills the main deep-learning families used in scRNA-seq analysis, helping retrieval connect a new task to the relevant representation-learning, generative, and predictive modeling paradigms.

[7] Defining and Benchmarking Open Problems in Single-Cell Analysis. **Nature Biotechnology (2025).**
Malte D. Luecken, Scott Gigante, . . . , Fabian J. Theis, Smita Krishnaswamy.

This paper formalizes open benchmark tasks and evaluation principles for single-cell analysis, which is directly useful when CellForge needs to frame a prediction problem and choose defensible metrics.

[8] Machine Learning for Perturbational Single-Cell Omics. **Cell Systems (2021).**
Yuge Ji, Mohammad Lotfollahi, F. Alexander Wolf, Fabian J. Theis.
This review lays out the core prediction tasks, datasets, and model classes in perturbational single-cell omics, making it a central retrieval paper for orienting architecture design around the actual problem taxonomy.

[9] Machine Learning to Dissect Perturbations in Complex Cellular Systems. **Computational and Structural Biotechnology Journal (2025).**
Pablo Monfort-Lanzas, Katja Rungger, Leonie Madersbacher, Hubert Hackl.
This review expands perturbation modeling beyond simple single-cell settings to broader cellular systems, helping retrieval connect CellForge to methods that capture complex responses, interactions, and experimental contexts.

[10] Quantifying the Effect of Experimental Perturbations at Single-Cell Resolution. **Nature Biotechnology (2021).**
Daniel B. Burkhardt, Jay S. Stanley, . . . , David van Dijk, Smita Krishnaswamy.
This work develops principled ways to measure perturbation effects at single-cell resolution, which is valuable for retrieval when the agent must reason about effect size, response structure, and evaluation targets.

[11] Reply to: Deeper Evaluation of a Single-Cell Foundation Model. **Nature Machine Intelligence (2024).**
Fan Yang, Fang Wang, . . . , Junzhou Huang, Jianhua Yao.
This reply sharpens the debate over how single-cell foundation models should be evaluated, making it useful for retrieval whenever CellForge needs to weigh claimed generalization against benchmark design and critique.

[12] Zero-Shot Evaluation Reveals Limitations of Single-Cell Foundation Models. **Genome Biology (2025).**
Kasia Z. Kedzierska, Lorin Crawford, Ava P. Amini, Alex X. Lu.
This paper documents substantial limits of zero-shot single-cell foundation models on benchmark tasks, giving CellForge concrete evidence that pretrained embeddings alone may not be sufficient for perturbation prediction.

Perturbation-response models.

[13] Cell Type Prioritization in Single-Cell Data. **Nature Biotechnology (2021).**
Michael A. Skinnider, Jordan W. Squair, . . . , Ariel J. Levine, Grégoire Courtine.
This paper turns perturbation effects into a cell-type ranking problem, giving CellForge a concrete example of how transcriptional signatures can be used to prioritize responsive populations rather than only predict average expression shifts.

[14] CellBox: Interpretable Machine Learning for Perturbation Biology with Application to the Design of Cancer Combination Therapy. **Cell Systems (2021).**
Bo Yuan, Ciyue Shen, . . . , John Ingraham, Chris Sander.
CellBox shows that dynamic, interpretable models can simulate multi-perturbation responses and support intervention design, which is directly relevant when CellForge retrieves mechanistic alternatives to purely black-box predictors.

[15] Decoding Heterogeneous Single-Cell Perturbation Responses. **Nature Cell Biology (2025).**
Bicna Song, Dingyu Liu, . . . , Danwei Huangfu, Wei Li.
This work emphasizes that perturbation effects are strongly cell-state dependent and heterogeneous across cells, making it useful when CellForge needs literature support for designs that model subpopulation-specific responses instead of a single pooled effect.

[16] Dissecting Cell Identity via Network Inference and in Silico Gene Perturbation. **Nature (2023).**
Kenji Kamimoto, Blerta Stringa, . . . , Lilianna Solnica-Krezel, Samantha A. Morris.
This paper infers regulatory networks and uses in silico perturbation to probe cell identity, offering CellForge a mechanistic template for retrieval when candidate models should incorporate gene-regulatory structure rather than rely only on direct supervised prediction.

[17] LangPert: LLM-Driven Contextual Synthesis for Unseen Perturbation Prediction. **ICLR 2025 Workshop on Machine Learning for Genomics Explorations.**
Kaspar Märtens, Marc Boubnovski Martell, Cesar A. Prada-Medina, Rory Donovan-Maiye.
LangPert is especially relevant to CellForge because it uses language-model-based contextual synthesis for unseen perturbation prediction, directly overlapping with the idea of using retrieved knowledge to propose models beyond the observed perturbation space.

[18] PerturbNet Predicts Single-Cell Responses to Unseen Chemical and Genetic Perturbations. **Molecular Systems Biology (2025).**
Hengshi Yu, Weizhou Qian, Yuxuan Song, Joshua D Welch.
PerturbNet targets generalization to unseen chemical and genetic interventions, making it a strong retrieval anchor for CellForge when the task requires architectures that extrapolate beyond perturbations present during training.

[19] Predicting Cellular Responses to Complex Perturbations in High-throughput Screens. **Molecular Systems Biology (2023).**
Mohammad Lotfollahi, Anna Klimovskaia Susmelj, . . . , David Lopez-Paz, Fabian J Theis.

This paper addresses compositional and high-order perturbation settings in large screening datasets, which is useful for CellForge when retrieved evidence needs to justify models for combinations rather than single interventions.

[20] Predicting Transcriptional Outcomes of Novel Multigene Perturbations with GEARS. **Nature Biotechnology (2024).**
Yusuf Roohani, Kexin Huang, Jure Leskovec.
GEARS demonstrates that gene-interaction structure can improve prediction for novel multigene perturbations, giving CellForge a concrete precedent for graph-aware architectures in combinatorial CRISPR settings.

[21] Predicting Transcriptional Responses to Novel Chemical Perturbations Using Deep Generative Model for Drug Discovery. **Nature Communications (2024).**
Xiaoning Qi, Lianhe Zhao, . . . , Shengyong Yang, Yi Zhao.
This paper extends deep generative perturbation modeling to novel chemical compounds, which helps CellForge retrieve evidence for architectures that bridge single-cell response prediction with drug-discovery-oriented chemical generalization.

[22] scGen Predicts Single-Cell Perturbation Responses. **Nature Methods (2019).**
Mohammad Lotfollahi, F. Alexander Wolf, Fabian J. Theis.
scGen is a foundational perturbation-prediction baseline that models responses as latent-space shifts in a conditional variational framework, so it remains an essential reference point for CellForge when comparing against standard generative approaches.

[23] scGNN Is a Novel Graph Neural Network Framework for Single-Cell RNA-Seq Analyses. **Nature Communications (2021).**
Juexin Wang, Anjun Ma, . . . , Qin Ma, Dong Xu.
Although broader than perturbation prediction alone, scGNN shows how graph neural networks can capture cell-cell and gene-gene structure in single-cell data, making it relevant when CellForge retrieves graph-based building blocks for perturbation models.

[24] Supervised training of conditional monge maps. **Advances in Neural Information Processing Systems (2022).**
Charlotte Bunne, Andreas Krause, Marco Cuturi.
This paper provides an optimal-transport formulation for learning conditional mappings between cellular states, which is valuable for CellForge retrieval when transport-based approaches are better suited than latent VAEs for modeling perturbation-induced state transitions.

Multimodal integration methods.

[25] Characterizing the Impacts of Dataset Imbalance on Single-Cell Data Integration. **Nature Biotechnology (2024).**
Hassaan Maan, Lin Zhang, . . . , Kieran R. Campbell, Bo Wang.
This paper shows how batch-size and cell-type imbalance can distort integration performance, which helps CellForge reason about when integration errors reflect data skew rather than model quality.

[26] Integrated Analysis of Multimodal Single-Cell Data. **Cell (2021).**
Yuhan Hao, Stephanie Hao, . . . , Peter Smibert, Rahul Satija.
This work introduces weighted nearest neighbors for joint RNA–protein and other multimodal analyses, making it a core reference for CITE-seq-style representation alignment.

[27] Integrative Single-Cell Analysis. **Nature Reviews Genetics (2019).**
Tim Stuart, Rahul Satija.
This review organizes the main computational strategies for batch correction, reference mapping, and cross-dataset integration, giving CellForge a high-level map of the integration design space.

[28] Joint Probabilistic Modeling of Single-Cell Multi-Omic Data with totalVI. **Nature Methods (2021).**
Adam Gayoso, Zoë Steier, . . . , Aaron Streets, Nir Yosef.
This paper presents a joint probabilistic model for paired RNA and protein measurements that denoises both modalities in a shared latent space, making it directly relevant for multimodal perturbation modeling.

[29] MultiVI: Deep Generative Model for the Integration of Multimodal Data. **Nature Methods (2023).**
Tal Ashuach, Mariano I. Gabitto, . . . , Michael I. Jordan, Nir Yosef.
This work extends deep generative integration to RNA–ATAC data with partially missing modalities, which is useful when CellForge must handle sparse or incompletely paired multi-omic measurements.

[30] MuSe-GNN: Learning Unified Gene Representation From Multimodal Biological Graph Data. **NeurIPS (2023).**
Tianyu Liu, Yuge Wang, Rex Ying, Hongyu Zhao.
This paper learns unified gene embeddings from multimodal biological graphs, offering a graph-based route for incorporating structured biological priors into multimodal prediction.

[31] Single-Cell Multimodal Prediction via Transformers. **CIKM (2023).**
Wenzhuo Tang, Hongzhi Wen, . . . , Hui Liu, Jiliang Tang.
This work applies transformer architectures to single-cell multimodal prediction, making it a useful reference for sequence-style fusion of cross-modal cellular features.

Foundation models and language-model interfaces.

[32] Cell2Sentence: Teaching Large Language Models the Language of Biology. **ICML (2024).**
Daniel Levine, Syed A. Rizvi, . . . , Rahul Madhav Dhodapkar, David Van Dijk.
This paper maps single-cell states into natural-language-like representations, making it a useful reference for how LLM interfaces can absorb cellular information without discarding biological structure.

[33] DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome. **Bioinformatics (2021).**
Yanrong Ji, Zhihan Zhou, Han Liu, Ramana V. Davuluri.
DNABERT established that transformer pretraining on genomic sequences learns transferable nucleotide representations, which motivates sequence-native foundation modeling in the CellForge retrieval set.

[34] GeneCompass: Deciphering Universal Gene Regulatory Mechanisms with a Knowledge-Informed X-Species Foundation Model. **Cell Research (2024).**
Xiaodong Yang, Guole Liu, . . . , Ge Yang, Xin Li.
GeneCompass combines large-scale pretraining with explicit biological knowledge to recover regulatory mechanisms across species, offering a concrete example of foundation models that encode gene regulation rather than only expression similarity.

[35] GeneGPT: Augmenting Large Language Models with Domain Tools for Improved Access to Biomedical Information. **Bioinformatics (2024).**
Qiao Jin, Yifan Yang, Qingyu Chen, Zhiyong Lu.
GeneGPT shows how an LLM can call biomedical tools and databases during reasoning, which directly informs CellForge's use of retrieval-augmented agents instead of a closed-book language model.

[36] GenSLMs: Genome-scale Language Models Reveal SARS-CoV-2 Evolutionary Dynamics. **The International Journal of High Performance Computing Applications (2023).**
Maxim Zvyagin, Alexander Brace, . . . , Venkatram Vishwanath, Arvind Ramanathan.
GenSLMs demonstrates that genome-scale language models can capture evolutionary regularities from massive sequence corpora, broadening the retrieval pool beyond transcriptomics to biologically meaningful foundation-model pretraining at scale.

[37] scGenePT: Is Language All You Need for Modeling Single-Cell Perturbations?. **bioRxiv (2024).**
Ana-Maria Istrate, Donghui Li, Theofanis Karaletsos.
scGenePT tests whether language-model-derived gene embeddings are sufficient for single-cell perturbation prediction, making it directly relevant to whether LLM-style priors help on CellForge's core task.

[38] scGPT: Toward Building a Foundation Model for Single-Cell Multi-Omics Using Generative AI. **Nature Methods (2024).**
Haotian Cui, Chloe Wang, . . . , Nan Duan, Bo Wang.
scGPT is a central reference for pretrained transformer modeling of single-cell multi-omics, providing a concrete baseline for how foundation models can represent cells, genes, and perturbational context in one framework.

[39] Simple and Effective Embedding Model for Single-Cell Biology Built from ChatGPT. **Nature Biomedical Engineering (2025).**
Yiqun Chen, James Zou.
This work shows that strong single-cell embeddings can be distilled from ChatGPT-derived supervision with a relatively simple pipeline, which is useful as a counterpoint to heavier bespoke foundation-model architectures.

[40] xTrimoPGLM: unified 100B-scale pre-trained transformer for deciphering the language of protein. **arXiv (2024).**
Bo Chen, Xingyi Cheng, . . . , Jie Tang, Le Song.
xTrimoPGLM extends large-scale biological language modeling to proteins at 100B-parameter scale, giving CellForge a retrieval anchor for foundation-model design patterns in molecular domains beyond transcriptomics.

Resources and tooling papers.

[41] Combinatorial Single-Cell CRISPR Screens by Direct Guide RNA Capture and Targeted Sequencing. **Nature Biotechnology (2020).**
Joseph M. Replogle, Thomas M. Norman, . . . , Jonathan S. Weissman, Britt Adamson.
This paper establishes a scalable protocol for directly capturing guide identities in combinatorial Perturb-seq experiments, which matters because many CellForge tasks depend on precisely annotated perturbation labels.

[42] Genome-Scale CRISPR-Cas9 Knockout and Transcriptional Activation Screening. **Nature Protocols (2017).**
Julia Joung, Silvana Konermann, . . . , Neville E. Sanjana, Feng Zhang.
This protocol paper details practical genome-scale CRISPR knockout and activation screening workflows, grounding retrieval in the experimental setup behind CRISPRi and CRISPRa datasets.

[43] Massively Multiplex Chemical Transcriptomics at Single-Cell Resolution. **Science (2020).**
Sanjay R. Srivatsan, José L. McFaline-Figueroa, . . . , Jay Shendure, Cole Trapnell.

This work introduces large-scale multiplexed chemical perturbation profiling at single-cell resolution, providing a canonical resource for drug-response prediction settings.

[44] SCANPY: Large-Scale Single-Cell Gene Expression Data Analysis. **Genome Biology (2018).**
F. Alexander Wolf, Philipp Angerer, Fabian J. Theis.
This paper provides the standard scalable analysis toolkit for preprocessing, neighborhood graphs, clustering, and visualization, so it anchors many downstream single-cell pipelines used around CellForge.

[45] scBaseCamp: An AI Agent-Curated, Uniformly Processed, and Continually Expanding Single Cell Data Repository. **bioRxiv (2025).**
Nicholas D. Youngblut, Christopher Carpenter, . . . , Hani Goodarzi, Yusuf H. Roohani.
This repository paper describes an agent-curated and uniformly processed single-cell data collection, which is relevant as a scalable source of standardized training and benchmarking datasets.

[46] scPerturb: Harmonized Single-Cell Perturbation Data. **Nature Methods (2024).**
Stefan Peidli, Tessa D. Green, . . . , Nils Blüthgen, Chris Sander.
This work harmonizes diverse single-cell perturbation datasets into a common resource, making it especially valuable for retrieval over comparable benchmarks and evaluation settings.

C Prompt templates and example outputs

This section collects the full prompt templates and representative output artifacts for the three CellForge team stages (Formulation, Ideation, and Execution), previously distributed across three separate team sections.

C.1 Task-preparation prompt structure

The Task Preparation team operates from three role-specific prompts: a Dataset Analyst for data characterisation, a Problem Investigator for prediction-task formulation, and a Baseline Assessor for reproducible reference benchmarks. Together they produce the task report that grounds the subsequent Ideation stage.

Dataset Analyst

"You are an expert in single-cell dataset analysis, specializing in evaluating single-cell perturbation data. Your task is to provide a comprehensive analysis of dataset characteristics and quality, focusing on experimental design, data properties, preprocessing recommendations, and quality assessment."
{decision_context}

{design_context}

1. Analyze Experimental Design:
- Evaluate experimental setup and biological conditions
- Assess sample size and biological replicates
- Review perturbation design and controls
- Analyze biological variability and batch effects
- Consider technical and biological validation

2. Assess Data Properties:
- Analyze data distribution and biological patterns
- Evaluate feature space and biological relevance
- Assess data quality and technical artifacts
- Identify biological and technical noise
- Consider data sparsity and missing values

3. Provide Preprocessing Recommendations:
- Suggest normalization strategies for biological data
- Recommend quality control procedures
- Propose feature selection methods
- Design batch effect correction approaches
- Plan biological validation steps

4. Conduct Quality Assessment:
- Evaluate data completeness and coverage
- Assess biological relevance and interpretability
- Analyze technical quality and reproducibility
- Consider biological validation requirements
- Identify potential limitations and biases

Task Description:
{task_description}

Dataset Information:

{json.dumps(dataset_info, indent=2)}

Relevant Literature:
{papers_context}

Please provide a comprehensive analysis in the following JSON format, incorporating decision support information:

Problem Investigator

"You are an expert in biological research problem analysis with extensive experience in designing computational solutions for biological applications. Your task is to provide a comprehensive investigation of the research problem and design a solution approach, focusing on problem definition, key challenges, research questions, and analysis methods."
1. Define Research Problem:
- Formally define the problem in biological context
- Identify key biological variables and their relationships
- Specify input-output mappings and biological constraints
- Define evaluation criteria with biological significance
- Establish success metrics with biological validation

2. Analyze Key Challenges:
- Identify biological and technical challenges
- Assess data quality and biological variability
- Evaluate computational complexity and scalability
- Consider biological interpretability requirements
- Address validation and reproducibility concerns

3. Formulate Research Questions:
- Define primary research questions with biological focus
- Identify key hypotheses to be tested
- Specify biological mechanisms to be investigated
- Outline experimental validation requirements
- Establish biological significance criteria

4. Design Analysis Methods:
- Propose computational approaches with biological relevance
- Design validation strategies with biological context
- Specify analysis pipelines with biological interpretation
- Plan experimental validation with biological controls
- Establish reproducibility standards with biological validation

Baseline Assessor

"You are an expert in deep learning model evaluation and benchmarking for single cell perturbation prediction, with extensive experience in assessing machine learning models for biological applications. Your task is to provide a comprehensive assessment of baseline models and evaluation strategies, focusing on model analysis, evaluation framework, performance analysis, and improvement suggestions."
{framework_context}

{guide_context}

Dynamic Analysis of Existing Methods (auto-extracted from retrieved literature):
{method_analysis}

1. Analyze Baseline Models for Single Cell Perturbation Prediction:
- Review existing deep learning methods for perturbation prediction
- Assess model architectures and their biological interpretability
- Evaluate implementation details and computational requirements
- Compare performance metrics and biological relevance
- Identify limitations and potential improvements

2. Design Evaluation Framework for Perturbation Prediction:
- Define evaluation metrics with biological significance
- Plan validation strategy considering biological variability
- Design test scenarios covering edge cases
- Establish benchmarks with state-of-the-art methods
- Consider biological relevance and interpretability

3. Conduct Performance Analysis:
- Analyze model performance across different perturbation types
- Compare with baselines and state-of-the-art methods
- Identify strengths/weaknesses in biological context
- Assess generalization to new cell types/conditions
- Evaluate robustness to biological and technical variations

4. Provide Improvement Suggestions:

- Identify weaknesses in current approaches
- Propose enhancements based on biological insights
- Suggest optimizations for better performance
- Address limitations in biological interpretation
- Plan future work with biological validation

Task Description:
{task_description}

Dataset Information:
{json.dumps(dataset_info, indent=2)}

Relevant Literature:
{papers_context}

Please provide a comprehensive assessment in the following JSON format:

Illustrative Task Preparation Output

Timestamp: "20260320_015255"
Task Description: "Predict post-CRISPRi gene expression in K562 cells from baseline expression and target gene identity ... Evaluate unseen perturbations, unseen cell contexts, and metrics including MSE, PCC, R^2, MSE_DE, PCC_DE, and R2_DE."
Dataset Info
Dataset Path: "./data/datasets/norman"
Dataset Name: "norman"
Data Type: "scRNA-seq"
Cell Line: "K562"
Perturbation Type: "CRISPRi"
Task Type: "gene_knockout"
Analysis Results
Dataset Analysis
Content
Experimental Design
Setup
Biological Conditions
- "K562 cell line under CRISPRi perturbation"
- "HEK293 cell l under CRISPRi perturbation"
- "A549 cell line under CRISPRi perturbation"
- ...

Experimental Protocol: "Cells treated with CRISPRi targeting specific genes under controlled conditions ..."
Controls
- "Unperturbed K562 cells"
- "Mock CRISPRi control"
- ...

Replicates: "At least 3 biological replicates per condition ..."
Perturbation
Design: "CRISPR interference targeting specific genes ..."
Timepoints
- "24h, 48h, and 72h post-perturbation"

Decision Support Considerations: "Ensure sufficient controls to distinguish biological effects from technical noise ..."
Samples
Size: "Sample size determined by power analysis ..."
Distribution: "Uniform distribution across replicates and conditions ..."
Validation
Biological
Methods
- "qRT-PCR"
- "Western Blot"
- ...

Technical
Methods
- "Sequencing depth analysis"
- "Technical replicates"
- ...

Data Properties
Distribution
Biological
Patterns
- "Clustered gene expression profiles indicating cell state"

Variability: "Notable variability across conditions and replicates ..."
Outliers
- "Cells deviating from expected expression profiles"

Technical
Patterns
- "Batch effects observed in expression profiles"

- **Artifacts**
  - "Low-quality reads, ambient RNA contamination"
  - ...
- **Features**
  - **Biological**: "Gene expression levels, gene ontology terms ..."
  - **Technical**: "Technical replicates, sequencing quality metrics ..."
- **Quality**
  - **Biological Metrics**
    - "Expression consistency across replicates"
    - "Statistical power of differential expression analysis"
  - **Technical Metrics**
    - "Sequencing depth, mapping quality scores"
- **Noise**
  - **Biological Sources**: "Natural biological variability, gene expression noise"
  - **Technical Sources**: "Sequencing errors, amplification biases"

Preprocessing
- **Normalization**
  - **Strategies**
    - "Log normalization, scaling by total counts ..."
  - **Quality Control**
    - "Outlier detection, filtering low-quality cells."
    - "Minimum gene counts, percentage of mitochondrial genes."
  - **Feature Selection**
    - "Differential expression analysis, variance filtering"
  - **Batch Correction**
    - "Combat, Harmony"
- **Validation**
  - **Biological**: "Functional assays, gene knockdown validation ..."
  - **Technical**: "Reproducibility checks, sequencing depth assessments ..."

Quality Assessment
- **Completeness**: "Coverage, missing data, mitigation, validation ..."
- **Relevance**: "Biological relevance, technical relevance, validation ..."
- **Reproducibility**: "Technical and biological reproducibility ..."
- **Limitations**: "Generalizability, batch effects, sequencing errors ..."

Confidence Score: 1.0
Timestamp: "2026-03-20T01:51:22.136787"
Metadata
- **Agent**: "Dataset Analyst"

Problem Investigation
Content
- **Problem Definition**
  - **Formal Definition**
    - **Biological Context**: "Predict gene expression profiles of K562 cells following CRISPRi perturbations ..."
    - **Input Output Mapping**: "Input: baseline expression + target gene(s); Output: post-perturbation expression ..."
    - **Evaluation Criteria**
      - "Accuracy of predicted gene expression profiles compared to observed data."
      - "Biological relevance of the predicted changes in gene expression."
      - ...
    - **Success Metrics**
      - "Mean Squared Error (MSE)"
      - "Pearson Correlation Coefficient (PCC)"
      - "R^2 (Coefficient of Determiation)"
      - "MSE for Differentially Expressed Genes (MSE_DE)"
      - "PCC for Differentially Expressed Genes (PCC_DE)"
      - "R^2 for Differentially Expressed Genes (R2_DE)"
  - **Key Variables**
    - **Biological**
      - "Gene expression levels"
      - "Target gene identity"
      - "Cell heterogeneity"
      - ...
    - **Technical**
      - "Computational models used"
      - "Data preprocessing techniques"
      - ...
  - **Scope**: "Focuses on K562 CRISPRi response while discussing broader applicability ..."
- **Key Challenges**
  - **Biological**: "Regulatory complexity, context-specific variability, confounders ..."
  - **Technical**: "High dimensionality, computational cost, interpretability trade-offs ..."
  - **Data Quality**: "Batch effects, missing data, cross-context inconsistency ..."
  - **Computational**: "Overfitting, long training time, scalability ..."
  - **Interpretability**: "Need explainable AI and researcher-facing outputs ..."
- **Research Questions**
  - **Primary**
    - **Questions**
      - "How accurately can we predict gene expression changes ... ?"
      - "What factors drive variability across biological contexts ... ?"

Hypotheses
- "Gene expression changes can be predicted with high accuracy ..."
- "Cellular context significantly affects perturbation response ..."

Secondary: "Off-target effects, perturbation strategy comparison ..."
Biological Mechanisms: "Gene regulatory networks, feedback loops, pathway cross-talk ..."
Experimental Validation: "CRISPRi perturbation, RNA-seq readout, baseline controls ..."
Analysis Methods
Computational Approaches: "Machine learning models, statistical models ..."
Validation Strategies: "Leave-one-out CV, bootstrap, external datasets ..."
Analysis Pipelines: "Preprocessing, model training, evaluation ..."
Experimental Validation: "RNA-seq, qPCR, comparative analysis ..."
Reproducibility: "Documentation, code/data sharing, FAIR reporting ..."
Confidence Score: 1.0
Timestamp: "2026-03-20T01:52:04.610853"
Metadata
Agent: "Problem Investigator"
Baseline Assessment
Content
Baseline Models
Literature Review
Existing Methods
Methods
- "scGPT"
- "PerturbNet"

Strengths: "Handles high-dimensional single-cell data effectively ..."
Limitations: "Large data requirement, substantial computational cost, limited interpretability ..."
Recommendations
- "Utilize scGPT for its generative capabilities ..."
- "Consider PerturbNet for unseen perturbation predictions ..."
- "Explore hybrid models ..."

Evaluation Framework
Metrics
Primary Metrics
- "Mean Squared Error (MSE)"
- "Pearson Correlation Coefficient (PCC)"

Secondary Metrics
- "R^2"
- "MSE for Differentially Expressed Genes (MSE_DE)"

Biological Metrics
- "PCC for Differentially Expressed Genes (PCC_DE)"
- "R^2 for Differentially Expressed Genes (R^2_DE)"

Validation Strategy: "K-fold cross-validation, biological validation, independent testing ..."
Benchmarks: "scGPT, PerturbNet, GeneCompass, DeepCell ..."
Performance Analysis
Model Performance: "Accuracy metrics, biological relevance, generalization, robustness ..."
Comparative Analysis: "Baseline comparison, SOTA comparison, biological comparison ..."
Strengths: "Advanced predictive capability, complex interaction modeling ..."
Weaknesses: "High computational cost, difficult interpretability ..."
Improvement Suggestions
Model Enhancements: "Attention mechanisms, hybrid models, transfer learning, data augmentation ..."
Implementation Optimizations: "Code refactoring, parallel processing, workflow automation ..."
Future Work: "Multi-omics integration, clinical translation, graph neural networks ..."
Confidence Score: 1.0
Timestamp: "2026-03-20T01:52:29.675735"
Metadata
Agent: "Baseline Assessor"
Refinement Comments: []
Final Recommendations
Model Selection
Recommended Models
Primary
- "scGPT"

Secondary
- "PerturbNet"

Implementation Priority
High Priority
- "Data preprocessing and quality control"

Medium Priority
- "Model training optimization"

Low Priority
- "Exploration of hybrid models"

C.2 Research-ideation prompt structure

The Research Ideation team uses a coordinated set of expert-selection and expert-discussion prompts to turn the task report into a model-design proposal. The prompt templates are shown below, followed by an illustrative integrated output artifact.

Expert Selector

"You are an expert coordinator for a scientific research team. Based on the task requirements, select the most appropriate experts to collaborate on this project."

Task Type: {task_type}
Task Analysis: {task_summary}

Available Experts:
{expert_profiles}

Requirements:
1. Select three expert roles plus one critic, covering the following core perspectives:
- Data Modeling (for single-cell data processing and pipeline design)
- Single-Cell Biology (for biological context and interpretation)
- Deep Learning (for model architecture and training)
2. Always include a Critic agent that scores proposals on scientific merit and technical feasibility.
3. Ensure the team's three expert roles have complementary skills covering all aspects of the task.

Please respond with only the names of the selected experts, one per line:

Expert Discussion: Proposal

"Generate context-aware prompt for expert discussion"

—

Task Type: {task_analysis['task_type']}
Dataset: {task_analysis['dataset']['name']} ({task_analysis['dataset'].get('type', 'Unknown type')})
Dataset Description: {task_analysis['dataset'].get('description', 'No description provided')}

Perturbations:
{self._format_perturbations(task_analysis['perturbations'])}

Cell Types: {', '.join(task_analysis['cell_types'])}

Objectives:
{self._format_list(task_analysis['objectives'])}

Constraints:
{self._format_list(task_analysis['constraints'])}

Evaluation Metrics:
{self._format_list(task_analysis['evaluation_metrics'])}

—

You are {expert_name}, an expert in {expert_domain}.

Your task is to design a state-of-the-art model architecture for single cell perturbation prediction that outperforms current baselines. Your expertise in {expert_domain} is crucial for this task.

TASK DETAILS:
{task_description}

DESIGN GOALS:
1. The architecture must be specifically optimized for {task_analysis['task_type']} in single-cell data
2. The design should incorporate domain-specific knowledge about cellular responses
3. The model should achieve better performance than current SOTA baselines
4. The architecture should be interpretable and biologically meaningful
5. The implementation should be computationally efficient and scalable

Your role is to contribute your expertise in {expert_domain} to ensure these goals are met.

—

Please propose a detailed architectural solution focusing on your domain expertise. Your proposal should aim to surpass current SOTA models in single-cell perturbation prediction.

Required sections:

1. Architecture Components:
- Detailed description of each component
- Configuration parameters and their justification
- How each component addresses specific challenges in single-cell data
- Integration points with other components

2. Biological Considerations:
- How the architecture handles cell type specificity
- Incorporation of pathway knowledge
- Mechanisms for capturing perturbation effects
- Biological constraints and their implementation

3. Technical Implementation:
- Detailed layer specifications
- Activation functions and their biological relevance
- Loss function design
- Training strategy recommendations

4. Performance Optimization:
- Specific techniques to outperform SOTA
- Computational efficiency considerations
- Scalability solutions
- Memory optimization strategies

5. Integration and Interpretability:
- Methods for model interpretation
- Visualization approaches
- Integration with existing biological knowledge
- Validation strategies

Format your response as a structured proposal with clear sections and specific technical details.

Expert Discussion: Feedback

"Please provide detailed feedback on the previous proposals, focusing on improving model performance beyond SOTA:"

1. Technical Analysis:
- Architectural design evaluation
- Component integration assessment
- Performance bottleneck identification
- Scalability analysis
- Memory efficiency review

2. Biological Relevance:
- Cell type specificity assessment
- Pathway integration evaluation
- Perturbation response mechanism analysis
- Biological constraint compliance
- Interpretability assessment

3. Implementation Feasibility:
- Technical complexity evaluation
- Resource requirements analysis
- Development timeline estimation
- Risk assessment
- Mitigation strategies

4. Improvement Suggestions:
- Specific architectural enhancements
- Performance optimization techniques
- Integration improvements
- Biological accuracy improvements
- Implementation recommendations

5. Comparative Analysis:
- Comparison with SOTA approaches
- Identification of competitive advantages
- Performance gap analysis
- Innovation assessment
- Potential impact evaluation

Be specific, technical, and constructive in your feedback. Focus on actionable improvements that can help surpass SOTA performance.

Expert Discussion: Critic

"Based on your domain expertise, ask critical questions about the current proposals:"

1. Technical Clarifications:
- Architecture design choices
- Implementation details
- Performance optimization strategies
- Integration mechanisms
- Scaling approaches

2. Biological Considerations:
- Cell type handling
- Pathway integration methods
- Perturbation response modeling
- Biological constraint implementation
- Validation approaches

3. Performance Questions:
- SOTA comparison methods
- Efficiency improvements
- Accuracy enhancements
- Scalability solutions
- Resource optimization

4. Integration Concerns:
- Component interactions
- Data flow
- Information sharing
- X-module dependencies
- System coherence

5. Validation and Testing:
- Performance metrics
- Biological validation
- Technical verification
- Quality assurance
- Benchmark comparisons

Frame your questions to drive the discussion toward concrete improvements and SOTA-surpassing solutions.

Expert Discussion: Judgment

"As a scientific critic, provide a comprehensive evaluation of the current proposals:"

1. Scientific Merit (Score 0.1–1.0):
- Theoretical foundation
- Methodological rigor
- Innovation level
- Technical sophistication
- Biological relevance

2. Technical Feasibility (Score 0.1–1.0):
- Implementation complexity
- Resource requirements
- Scalability potential
- Performance expectations
- Maintenance considerations

3. Biological Validity (Score 0.1–1.0):
- Cell type handling
- Pathway integration
- Perturbation modeling
- Constraint compliance
- Interpretability

4. SOTA Comparison (Score 0.1–1.0):
- Performance potential
- Innovation aspects
- Technical advantages
- Biological accuracy
- Overall competitiveness

5. Implementation Strategy (Score 0.1–1.0):
- Development approach
- Resource allocation

- Timeline realism
- Risk management
- Quality assurance

Provide a detailed justification for each score and an overall score (0.1–1.0) weighted across all categories.
Focus on potential for surpassing SOTA performance in single-cell perturbation prediction.

Illustrative Research Ideation Output

Timestamp: "2026-03-20 01:58:27"
Model Architecture
Encoder: type "transformer"; layers 4; heads 8; dimension 512; activation "gelu"
Perturbation Encoder: type "mlp"; layers 3; dimensions [256, 128, 64]; activation "swish"
Cross Attention: heads 8; dimension 512; dropout 0.1
Decoder: type "mlp"; layers 3; dimensions [256, 512, 1024]; activation "swish"
Data Processing
Quality Control
Cell Filtering: min genes 200; max genes 6000; min counts 1000; max counts 50000; max mito percent 20
Gene Filtering: min cells 3; min counts 10
Normalization: method "log1p"; target sum 10000; regression vars ["total_counts", "pct_counts_mt"]
Batch Correction: method "harmony"; theta 2; max iterations 20
Training Strategy
Optimizer: type "adamw"; learning rate 0.0003; weight decay 0.01
Scheduler: type "cosine_annealing"; T_0 10; eta_min 1e-06
Training: batch size 64; gradient clip 1.0; early stopping patience 15; min delta 0.0001
Evaluation Metrics
Loss Functions: reconstruction "mse"; perturbation "bce"; biological "pathway_consistency"
Implementation Details
Framework: "PyTorch"
Hardware Requirements: GPU "NVIDIA GPU with 8GB+ VRAM"; RAM "32GB+ system RAM"; storage "100GB+ SSD"
Dependencies: "torch>=1.12.0", "scanpy>=1.9.0", "numpy>=1.21.0", "scikit-learn>=1.0.0"
Code Structure: data loader, modular model, checkpointed training, comprehensive evaluation pipeline
Biological Validation
Key Theoretical Foundations
Baseline Regulatory Info: assumption "Baseline contains sufficient regulatory info"; validation "Ablation study (masking key TFs)"; biological basis "Central dogma of molecular biology"
Dose Response Continuity: validation "Dose interpolation experiments"; biological basis "Ligand-receptor binding kinetics"
Pathway Modularity: validation "Pathway enrichment analysis"; biological basis "Known biological pathway organization"
Latent Space Smoothness: validation "UMAP visualization"; biological basis "Waddington epigenetic landscape concept"
Adaptive Mechanisms
Dose Specific Attention: "Higher attention weights to stress pathways at toxic doses"
Cell Type Gating: "Cell-type specific pathway modulation"
Chemical Class Priors: "Initialize embeddings based on chemical class"
Computational Optimization
Training Techniques
Progressive Dose Sampling
epoch_0_10: "uniform(0, 1µM)"
epoch_10_20: "log_normal(1µM-10µM)"
epoch_20+: "full_range(0-100µM)"
Regularization Methods
Pathway Sparsity: type "l1"; lambda 0.01; target "pathway_weights"
Dose Consistency: type "custom"; formula "L_consist = E[||f(x,d1) - f(f(x,d1),d2-d1)||²]"
Parallelization
Gene Cluster Parallelism
chr1: GPU0; genes ["EGFR", "MYC"]
chr17: GPU1; genes ["TP53", "BRCA1"]
others: GPU2; genes "Housekeeping"
Model Assumptions
Data Assumptions: sparse/imbalanced data, batch effects, technical noise; solutions include advanced sampling, Harmony, and robust preprocessing
Computational Assumptions: dimensionality reduction, transfer learning, optimized implementations
Extensions
Drug Discovery
Virtual Screening: predict expression profiles for novel compounds
Drug Repurposing: identify new indications based on expression patterns
Side Effects: analyze off-target expression changes
Precision Medicine
Patient Specific: use baseline expression for personalized predictions
Drug Combinations: predict synergistic effects
Biomarkers: identify sensitivity markers
Model Extensions
Multi Modal: integrate proteomics and metabolomics
Time Series: model temporal drug effects
Mechanistic: combine with pathway models
Mermaid Diagrams

System Overview
Title: "Research Framework Architecture"
Description: complete framework for biological perturbation analysis using single-cell data
Mermaid Code: "graph TD ..."
Modules
Data Input Layer: raw single-cell data and metadata; required fields ["counts", "var_names", "obs_names"]
Data Processing Layer: quality control, normalization, feature selection via scanpy methods
Model Architecture Layer: encoder, decoder, biological constraints
Training Layer: optimizer, loss functions, validation procedures
Evaluation Layer: technical metrics, biological metrics, expert/literature validation
Output Layer: predictions, biological insights, applications
Detailed Architecture
Title: "Detailed Component Architecture"
Description: encoder, decoder, training, and evaluation components; mermaid code "graph LR ..."
Framework Overview
Title: "Biological Perturbation Analysis Framework"
Key Features: multi-expert integration; RAG-enhanced retrieval; biological constraints; comprehensive metrics; scalable architecture
Supported Tasks: "gene_knockout", "drug_perturbation", "cytokine_stimulation"
Output Formats: "markdown", "json", "mermaid", "png"
Expert Domains: Data Engineering, Single Cell Biology, Deep Learning, Molecular Biology, Bioinformatics, Statistics, Drug Discovery, Gene Regulation, Experimental Design, Validation, Baseline Assessment, Training, Drug Response, Pathway Analysis, Cell Communication, Omics Integration
Task Wise Plan
T01: objective "Predict gene expression responses after perturbation"
Scope: task type "gene_knockout"; dataset "norman" ("scRNA-seq"); cell types ["K562"]
Implementation Contract: inputs baseline expression, perturbation identity, cell context features; outputs predicted expression profile and task-level metrics
Evaluation Metrics: "MSE", "PCC", "R2", "MSE_DE", "PCC_DE", "R2_DE", ...
Success Criteria: metrics reported; reproducible from config/seed
Design Refs: repeats architecture, preprocessing, training strategy, and implementation details above
T02: objective "Generalize to unseen perturbations and unseen cell contexts"
Scope: same dataset/task context with emphasis on OOD generalization
Implementation Contract: same inputs, outputs, and metric set
Design Refs: same transformer + MLP architecture, Harmony preprocessing, AdamW training, and PyTorch implementation stack

C.3 Experiment-execution prompt structure

The Experiment Execution team converts the approved design into runnable code and training outputs. The Code Generator prompt below is the main prompt used in this stage; an illustrative example output follows.

Code Generator

"You are a computational biologist and expert Python developer. Your task is to write correct, efficient, and well-documented code based on the provided research plan for single-cell perturbation prediction."

IMPORTANT REQUIREMENTS:
1. Write all code in a single file named `result.py`
2. Follow the research plan specifications exactly
3. Implement proper data processing, model architecture, and training pipeline
4. Use appropriate libraries (scanpy, torch, numpy, pandas, etc.)
5. Include comprehensive error handling and validation
6. Add detailed comments explaining biological and computational logic

EXECUTION CONSTRAINTS:
- Do not run shell commands or tools.
- Do not inspect the local filesystem.
- Produce code directly from the provided research plan.

Output only the final Python code. If you include markdown, wrap exactly one Python block.

Illustrative Experiment Execution Output

```
import json
import logging
import random
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Dict, List, Optional, Sequence, Tuple

import numpy as np
import scanpy as sc
import torch
import torch.nn as nn
import torch.nn.functional as F
from anndata import AnnData
```

```
from sklearn.preprocessing import LabelEncoder
from torch.utils.data import DataLoader, Dataset


logging.basicConfig(level=logging.INFO)
LOGGER = logging.getLogger("CellForge")
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def set_seed(seed: int = 42) -> None:
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


@dataclass
class QCCfg:
    min_genes: int = 200
    max_genes: int = 6000
    min_counts: int = 1000
    max_counts: int = 50000
    max_mito_percent: float = 20.0


@dataclass
class TrainingCfg:
    batch_size: int = 64
    gradient_clip: float = 1.0
    patience: int = 15
    min_delta: float = 1e-4
    max_epochs: int = 200
    lr: float = 3e-4
    weight_decay: float = 0.01


def preprocess_data(adata: AnnData, qc_cfg: QCCfg) -> AnnData:
    adata.var["MT"] = adata.var_names.str.upper().str.startswith("MT-")
    sc.pp.calculate_qc_metrics(adata, qc_vars=["MT"], inplace=True)
    sc.pp.filter_cells(adata, min_genes=qc_cfg.min_genes)
    sc.pp.filter_cells(adata, min_counts=qc_cfg.min_counts)
    adata = adata[adata.obs["pct_counts_mt"] < qc_cfg.max_mito_percent].copy()
    sc.pp.normalize_total(adata, target_sum=10000)
    sc.pp.log1p(adata)
    sc.pp.highly_variable_genes(adata, flavor="seurat_v3", n_top_genes=4000)
    return adata[:, adata.var["highly_variable"]].copy()


class SingleCellPerturbationDataset(Dataset):
    def __init__(
        self,
        adata: AnnData,
        gene_list: Sequence[str],
        perturbation_key: str = "perturbation",
        dose_key: str = "dose",
        condition_key: str = "condition",
    ):
        self.expression = (
            adata[:, gene_list].X.toarray()
            if hasattr(adata.X, "toarray")
            else adata[:, gene_list].X
        )
        self.perturbations = adata.obs[perturbation_key].astype(str).values
        self.doses = adata.obs[dose_key].astype(float).values
        self.conditions = adata.obs[condition_key].astype(str).values
        self.perturb_encoder = LabelEncoder().fit(self.perturbations)
        self.condition_encoder = LabelEncoder().fit(self.conditions)

    def __len__(self) -> int:
        return len(self.expression)

    def __getitem__(self, idx: int) -> Dict[str, torch.Tensor]:
        expr = torch.tensor(self.expression[idx], dtype=torch.float32)
        return {
            "baseline": expr,
            "target": expr,
            "perturbation": torch.tensor(
                self.perturb_encoder.transform([self.perturbations[idx]])[0]
            ),
            "condition": torch.tensor(
                self.condition_encoder.transform([self.conditions[idx]])[0]
            ),
            "dose": torch.tensor(self.doses[idx], dtype=torch.float32),
        }


class PerturbationPredictor(nn.Module):
    def __init__(self, input_dim: int, num_perturbations: int, num_conditions: int):
        super().__init__()
        self.gene_encoder = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.GELU(),
            nn.Linear(512, 512),
        )
        self.perturb_embed = nn.Embedding(num_perturbations, 256)
        self.condition_embed = nn.Embedding(num_conditions, 256)
        self.cross_attn = nn.MultiheadAttention(
            embed_dim=512, num_heads=8, dropout=0.1, batch_first=True
        )
        self.decoder = nn.Sequential(
            nn.Linear(512, 256),
            nn.SiLU(),
            nn.Linear(256, 512),
```

```
            nn.SiLU(),
            nn.Linear(512, input_dim),
        )

    def forward(
        self,
        baseline: torch.Tensor,
        perturbation: torch.Tensor,
        condition: torch.Tensor,
        dose: torch.Tensor,
    ) -> torch.Tensor:
        gene_latent = self.gene_encoder(baseline).unsqueeze(1)
        pert_latent = torch.cat(
            [
                self.perturb_embed(perturbation),
                self.condition_embed(condition),
            ],
            dim=-1,
        ).unsqueeze(1)
        fused, _ = self.cross_attn(gene_latent, pert_latent, pert_latent)
        return self.decoder(fused.squeeze(1))


def compute_metrics(pred: np.ndarray, target: np.ndarray) -> Dict[str, float]:
    mse = float(np.mean((pred - target) ** 2))
    pearson = float(np.corrcoef(pred.reshape(-1), target.reshape(-1))[0, 1])
    r2 = float(1 - np.sum((target - pred) ** 2) / (np.sum((target - target.mean()) ** 2) + 1e-8))
    return {"MSE": mse, "Pearson": pearson, "R2": r2}


@dataclass
class EarlyStopping:
    patience: int
    min_delta: float
    best_loss: float = field(default_factory=lambda: float("inf"))
    counter: int = 0

    def step(self, loss: float) -> bool:
        if loss + self.min_delta < self.best_loss:
            self.best_loss = loss
            self.counter = 0
            return False
        self.counter += 1
        return self.counter >= self.patience


class Trainer:
    def __init__(self, model: nn.Module, train_loader: DataLoader, val_loader: DataLoader, cfg: TrainingCfg):
        self.model = model.to(DEVICE)
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.cfg = cfg
        self.optim = torch.optim.AdamW(
            self.model.parameters(), lr=cfg.lr, weight_decay=cfg.weight_decay
        )
        self.early_stopping = EarlyStopping(cfg.patience, cfg.min_delta)

    def train_epoch(self) -> float:
        self.model.train()
        total_loss = 0.0
        for batch in self.train_loader:
            batch = {k: v.to(DEVICE) for k, v in batch.items()}
            pred = self.model(
                batch["baseline"],
                batch["perturbation"],
                batch["condition"],
                batch["dose"],
            )
            loss = F.mse_loss(pred, batch["target"])
            self.optim.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), self.cfg.gradient_clip)
            self.optim.step()
            total_loss += loss.item()
        return total_loss / len(self.train_loader)

    def evaluate(self) -> Tuple[float, Dict[str, float]]:
        self.model.eval()
        preds, targets, losses = [], [], []
        with torch.no_grad():
            for batch in self.val_loader:
                batch = {k: v.to(DEVICE) for k, v in batch.items()}
                pred = self.model(
                    batch["baseline"],
                    batch["perturbation"],
                    batch["condition"],
                    batch["dose"],
                )
                losses.append(F.mse_loss(pred, batch["target"]).item())
                preds.append(pred.cpu().numpy())
                targets.append(batch["target"].cpu().numpy())
        return float(np.mean(losses)), compute_metrics(
            np.concatenate(preds), np.concatenate(targets)
        )

    def fit(self) -> Dict[str, Any]:
        history = {"train_loss": [], "val_loss": [], "metrics": []}
        for epoch in range(self.cfg.max_epochs):
            train_loss = self.train_epoch()
            val_loss, metrics = self.evaluate()
            history["train_loss"].append(train_loss)
            history["val_loss"].append(val_loss)
            history["metrics"].append(metrics)
```

```
            LOGGER.info(
                "Epoch %d | Train %.4f | Val %.4f | Metrics %s",
                epoch,
                train_loss,
                val_loss,
                metrics,
            )
            if self.early_stopping.step(val_loss):
                break
        return history


def run_pipeline(data_path: str) -> Dict[str, Any]:
    set_seed(42)
    adata = sc.read_h5ad(data_path)
    adata = preprocess_data(adata, QCCfg())
    genes = list(adata.var_names)
    dataset = SingleCellPerturbationDataset(adata, genes)
    indices = np.arange(len(dataset))
    np.random.shuffle(indices)
    split = int(0.8 * len(indices))
    train_loader = DataLoader(
        torch.utils.data.Subset(dataset, indices[:split]),
        batch_size=64,
        shuffle=True,
    )
    val_loader = DataLoader(
        torch.utils.data.Subset(dataset, indices[split:]),
        batch_size=64,
        shuffle=False,
    )
    model = PerturbationPredictor(
        input_dim=len(genes),
        num_perturbations=len(set(adata.obs["perturbation"])),
        num_conditions=len(set(adata.obs["condition"])),
    )
    trainer = Trainer(model, train_loader, val_loader, TrainingCfg())
    history = trainer.fit()
    return {
        "history": history,
        "biological_validation": {
            "latent_space_smoothness": ...,
            "pathway_modularity_assertion": ...,
            "dose_consistency_check": ...,
        },
    }


def main() -> None:
    import argparse

    parser = argparse.ArgumentParser(description="Single-cell perturbation predictor")
    parser.add_argument("--data", required=True, help="Path to input AnnData file (.h5ad)")
    args = parser.parse_args()
    outputs = run_pipeline(args.data)
    LOGGER.info("Training complete. Final metrics: %s", outputs["history"]["metrics"][-1])
    LOGGER.info("Additional checks: %s", outputs["biological_validation"])


if __name__ == "__main__":
    main()

# ... additional modules for Harmony batch correction, pathway consistency loss,
# ... progressive dose sampling, checkpointing, and autoresearch refinements.
```

D Generated model implementations

The listings below reproduce the Python training scripts that CellForge actually wrote and ran for each of the six benchmark datasets. Each listing is the final `result.py` produced by the Execution stage for the canonical run whose architecture is named in the main text; the scripts handle data loading, the dataset-specific architecture, the training loop with five-fold cross-validation, and the evaluation pipeline. No post-hoc editing has been applied. Long lines are wrapped for typesetting; file boundaries preserve the as-run source.

D.1 *CPA-X* for Adamson CRISPRi (scRNA-seq)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
```

```
import time
from datetime import datetime
import argparse
import warnings
from typing import Dict, List, Tuple, Optional, Union
import random
warnings.filterwarnings('ignore')

def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")

def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
scaler = None
common_genes_info = None

class ControlPerturbedDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', scaler=None,
                 augment=False, is_train=True, common_genes_info=None, train_control_baseline=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.augment = augment
        self.is_train = is_train
        self.common_genes_info = common_genes_info
        self.train_control_baseline = train_control_baseline
        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X
        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)
        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)
        if not is_log1p_already:
            data = np.log1p(data)
        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            if not is_standardized_already:
                data = self.scaler.transform(data)
        data = np.clip(data, -10, 10)
        data = data / 10.0
        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self._create_control_perturbed_pairs()
        print_log(f"Dataset created: {len(self.pairs)} pairs, {self.expression_data.shape[1]} features")

    def _create_control_perturbed_pairs(self):
        perturbation_col = self.adata.obs[self.perturbation_key].astype(str)
        negctrl_mask = perturbation_col.str.contains('NegCtrl', case=False, na=False)
        lower_labels = perturbation_col.str.lower()
        exact_control_mask = (lower_labels == 'control')
        other_control_mask = perturbation_col.isin(['Control', 'ctrl', 'Ctrl', 'CONTROL'])
        contains_control_mask = perturbation_col.str.contains('control|ctrl', case=False, na=False)
        control_mask = exact_control_mask | negctrl_mask | other_control_mask | contains_control_mask
        control_indices = np.where(control_mask)[0]
        perturbed_indices = np.where(~control_mask)[0]
        print_log(f"Control samples: {len(control_indices)}")
        print_log(f"Perturbed samples: {len(perturbed_indices)}")
        if len(control_indices) == 0:
            unique_perturbations = perturbation_col.unique()[:10]
            print_log(f"DEBUG: Sample perturbation values: {unique_perturbations}")
            print_log("DEBUG: Attempting to find control samples with alternative patterns...")
        self.perturbation_names = list(perturbation_col.unique())
        self.perturbations = pd.get_dummies(perturbation_col).values.astype(np.float32)
        print_log(f"Perturbation dimension: {self.perturbations.shape[1]}")
        if len(control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
```

```
                    "Please ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                if self.train_control_baseline is None:
                    raise ValueError(
                        "CRITICAL ERROR: Test set has no control samples AND no training control baseline provided! "
                        "This is required for unseen perturbation prediction. "
                        "Please ensure training set has control samples."
                    )
                else:
                    print_log("=" * 60)
                    print_log("Test set: No control samples found in test data.")
                    print_log("Using training set control baseline (CORRECT for unseen perturbation task).")
                    print_log("=" * 60)
                    self.avg_baseline = self.train_control_baseline
                    self.use_avg_baseline = True
        else:
            print_log(f"Using {len(control_indices)} true control samples as baseline")
            control_baseline = np.mean(self.expression_data[control_indices], axis=0)
            self.control_baseline = control_baseline
            self.use_avg_baseline = False
            self.avg_baseline = None
        self.pairs = []
        for i, perturbed_idx in enumerate(perturbed_indices):
            if self.use_avg_baseline:
                baseline_idx = -1
            else:
                available_controls = control_indices[control_indices != perturbed_idx]
                if len(available_controls) > 0:
                    if self.is_train:
                        baseline_idx = np.random.choice(available_controls)
                    else:
                        baseline_idx = available_controls[i % len(available_controls)]
                else:
                    if self.is_train:
                        baseline_idx = np.random.choice(control_indices)
                    else:
                        baseline_idx = control_indices[i % len(control_indices)]
            self.pairs.append({
                'baseline_idx': baseline_idx,
                'perturbed_idx': perturbed_idx,
                'perturbation': self.perturbations[perturbed_idx]
            })
        if len(control_indices) > 0 and len(perturbed_indices) > 0:
            for i, control_idx in enumerate(control_indices):
                if self.is_train:
                    target_idx = np.random.choice(perturbed_indices)
                else:
                    target_idx = perturbed_indices[i % len(perturbed_indices)]
                self.pairs.append({
                    'baseline_idx': control_idx,
                    'perturbed_idx': target_idx,
                    'perturbation': self.perturbations[target_idx]
                })
        elif len(control_indices) > 1:
            for i, control_idx in enumerate(control_indices):
                other_controls = control_indices[control_indices != control_idx]
                if len(other_controls) > 0:
                    if self.is_train:
                        baseline_idx = np.random.choice(other_controls)
                    else:
                        baseline_idx = other_controls[i % len(other_controls)]
                    self.pairs.append({
                        'baseline_idx': baseline_idx,
                        'perturbed_idx': control_idx,
                        'perturbation': self.perturbations[control_idx]
                    })
        print_log(f"Total pairs created: {len(self.pairs)}")

    def __len__(self):
        return len(self.pairs)

    def __getitem__(self, idx):
        pair = self.pairs[idx]
        if pair['baseline_idx'] == -1:
            x_control = self.avg_baseline.copy()
        else:
            x_control = self.expression_data[pair['baseline_idx']]
        x_perturbed = self.expression_data[pair['perturbed_idx']]
        pert = pair['perturbation']
        if pair['baseline_idx'] != -1 and pair['baseline_idx'] == pair['perturbed_idx']:
            print_log(f"WARNING: Found self-pair (baseline_idx == perturbed_idx). This should not happen!")
            if hasattr(self, 'avg_baseline') and self.avg_baseline is not None:
                x_control = self.avg_baseline.copy()
            else:
                x_control = self.expression_data[0].copy()
            x_perturbed = self.expression_data[pair['perturbed_idx']]
            pert = pair['perturbation']
        x_perturbed_original = self.original_expression_data[pair['perturbed_idx']]
        x_target_abs = x_perturbed_original
        if self.augment and self.is_train:
            noise = np.random.normal(0, 0.05, x_control.shape)
            x_control = x_control + noise
            mask = np.random.random(x_control.shape) > 0.05
            x_control = x_control * mask
        return (torch.FloatTensor(x_control),
                torch.FloatTensor(pert),
                torch.FloatTensor(x_target_abs))

    def get_pert_dim(self):
        return self.perturbations.shape[1]

class PerturbationEncoder(nn.Module):
    def __init__(self, pert_dim, hidden_dim, dropout=0.1):
        super().__init__()
```

```
        self.hidden_dim = hidden_dim
        self.pert_embedding = nn.Linear(pert_dim, hidden_dim)
        self.encoder = nn.Sequential(
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )

    def forward(self, pert):
        pert_emb = self.pert_embedding(pert)
        output = self.encoder(pert_emb)
        return output

class StatePerturbationCrossAttention(nn.Module):
    def __init__(self, state_dim, pert_dim, hidden_dim, n_heads=8, dropout=0.1):
        super().__init__()
        assert hidden_dim % n_heads == 0, "hidden_dim must be divisible by n_heads"
        self.n_heads = n_heads
        self.head_dim = hidden_dim // n_heads
        self.hidden_dim = hidden_dim
        self.state_proj = nn.Linear(state_dim, hidden_dim)
        self.query_proj = nn.Linear(hidden_dim, hidden_dim)
        self.key_proj = nn.Linear(hidden_dim, hidden_dim)
        self.value_proj = nn.Linear(hidden_dim, hidden_dim)
        self.output_proj = nn.Linear(hidden_dim, hidden_dim)
        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(hidden_dim)
        self.scale = self.head_dim ** -0.5

    def forward(self, z_state, p_emb):
        batch_size = z_state.size(0)
        z_state_proj = self.state_proj(z_state).unsqueeze(1)
        p_emb_expanded = p_emb.unsqueeze(1)
        residual = z_state_proj
        Q = self.query_proj(z_state_proj).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        K = self.key_proj(p_emb_expanded).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        V = self.value_proj(p_emb_expanded).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        scores = torch.matmul(Q, K.transpose(-2, -1)) * self.scale
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)
        attended = torch.matmul(attn_weights, V)
        attended = attended.transpose(1, 2).contiguous().view(batch_size, 1, self.hidden_dim)
        output = self.output_proj(attended)
        output = self.layer_norm(output + residual)
        return output.squeeze(1)

class GALAModel(nn.Module):
    def __init__(self, input_dim, output_dim, pert_dim, hidden_dim=256, n_layers=2, n_heads=8,
                 dropout=0.1, attention_dropout=0.1, ffn_dropout=0.1, activation='gelu'):
        super(GALAModel, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.pert_dim = pert_dim
        self.hidden_dim = hidden_dim
        self.n_heads = n_heads
        self.expression_encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.pert_encoder = PerturbationEncoder(
            pert_dim=pert_dim,
            hidden_dim=hidden_dim,
            dropout=attention_dropout
        )
        self.cross_attention = StatePerturbationCrossAttention(
            state_dim=hidden_dim,
            pert_dim=hidden_dim,
            hidden_dim=hidden_dim,
            n_heads=n_heads,
            dropout=attention_dropout
        )
        mlp_layers = []
        for _ in range(n_layers):
            mlp_layers.extend([
                nn.Linear(hidden_dim, hidden_dim * 2),
                nn.LayerNorm(hidden_dim * 2),
                nn.GELU() if activation == 'gelu' else nn.ReLU(),
                nn.Dropout(ffn_dropout),
                nn.Linear(hidden_dim * 2, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU() if activation == 'gelu' else nn.ReLU(),
                nn.Dropout(ffn_dropout)
            ])
        self.mlp = nn.Sequential(*mlp_layers) if mlp_layers else nn.Identity()
        self.output_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, output_dim)
        )
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
```

```
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)

    def forward(self, x_control, pert):
        assert x_control.dim() == 2 and x_control.size(1) == self.input_dim, \
            f"Expected x_control shape [B, {self.input_dim}], got {x_control.shape}"
        z_state = self.expression_encoder(x_control)
        p_emb = self.pert_encoder(pert)
        h_int = self.cross_attention(z_state, p_emb)
        h_int = h_int + z_state
        x_trans = self.mlp(h_int)
        output = self.output_head(x_trans)
        return output

def train_model(model, train_loader, optimizer, scheduler, device, max_pert_dim=None):
    model.train()
    total_loss = 0
    for batch in tqdm(train_loader, desc="Training"):
        x_control, pert, x_target_abs = batch
        x_control, pert, x_target_abs = x_control.to(device), pert.to(device), x_target_abs.to(device)
        if max_pert_dim is not None and pert.shape[1] < max_pert_dim:
            pad_size = max_pert_dim - pert.shape[1]
            pert = F.pad(pert, (0, pad_size), mode='constant', value=0)
        optimizer.zero_grad()
        output = model(x_control, pert)
        loss = F.mse_loss(output, x_target_abs)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()
        total_loss += loss.item()
    return total_loss / len(train_loader)

def calculate_detailed_metrics(pred, true, control_baseline=None, perturbations=None):
    n_samples, n_genes = true.shape
    mse = np.mean((pred - true) ** 2)
    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0
    true_mean_vector = np.mean(true, axis=0)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_vector) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0
    if control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) / (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
                    pred_de_flat = pred_de.flatten()
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_vector = np.mean(true_de, axis=0)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
```

```
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                mse_de = np.mean((pred_de - true_de) ** 2)
                pred_de_flat = pred_de.flatten()
                true_de_flat = true_de.flatten()
                if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                    pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0
                true_de_mean_vector = np.mean(true_de, axis=0)
                ss_res_de = np.sum((true_de - pred_de) ** 2)
                ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                if ss_tot_de > 1e-10:
                    r2_de = 1.0 - (ss_res_de / ss_tot_de)
                else:
                    r2_de = 0.0
                if np.isnan(r2_de):
                    r2_de = 0.0
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }

def evaluate_model(model, test_loader, device, max_pert_dim=None, test_dataset=None):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    all_controls = []
    all_perts = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            x_control, pert, x_target_abs = batch
            x_control, pert, x_target_abs = x_control.to(device), pert.to(device), x_target_abs.to(device)
            if max_pert_dim is not None and pert.shape[1] < max_pert_dim:
                pad_size = max_pert_dim - pert.shape[1]
                pert = F.pad(pert, (0, pad_size), mode='constant', value=0)
            output = model(x_control, pert)
            loss = F.mse_loss(output, x_target_abs)
            total_loss += loss.item()
            all_targets.append(x_target_abs.cpu().numpy())
            all_predictions.append(output.cpu().numpy())
            all_controls.append(x_control.cpu().numpy())
            all_perts.append(pert.cpu().numpy())
    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_controls = np.concatenate(all_controls, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)
    r2 = r2_score(all_targets, all_predictions)
    if np.isnan(r2):
        r2 = 0.0
    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pearson = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pearson):
            pearson = 0.0
    else:
        pearson = 0.0
    if test_dataset is not None and hasattr(test_dataset, 'control_baseline') and test_dataset.control_baseline is not None and not (hasattr(test_dataset, 'use_avg_baseline') and
        test_dataset.use_avg_baseline):
        control_baseline = test_dataset.control_baseline.reshape(1, -1) if test_dataset.control_baseline.ndim == 1 else test_dataset.control_baseline
    elif test_dataset is not None and hasattr(test_dataset, 'use_avg_baseline') and test_dataset.use_avg_baseline:
        control_baseline = None
    else:
        unique_controls = np.unique(all_controls, axis=0)
        if len(unique_controls) > 0:
            control_baseline = all_controls if len(all_controls) > 0 else None
        else:
            control_baseline = None
    detailed_metrics = calculate_detailed_metrics(all_predictions, all_targets, control_baseline=control_baseline, perturbations=all_perts)
    return {
        'loss': total_loss / len(test_loader),
        'r2': r2,
        'pearson': pearson,
        'MSE': detailed_metrics['MSE'],
        'PCC': detailed_metrics['PCC'],
        'R2': detailed_metrics['R2'],
        'MSE_DE': detailed_metrics['MSE_DE'],
        'PCC_DE': detailed_metrics['PCC_DE'],
        'R2_DE': detailed_metrics['R2_DE']
    }
```

```
def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(list(set(train_dataset.perturbation_names + test_dataset.perturbation_names)))
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []
    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))
    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(
        columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)
    for i, pair in enumerate(train_dataset.pairs):
        pert_idx = pair['perturbed_idx']
        train_dataset.pairs[i]['perturbation'] = train_dataset.perturbations[pert_idx]
    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(np.float32)
        for i, pair in enumerate(test_dataset.pairs):
            pert_idx = pair['perturbed_idx']
            test_dataset.pairs[i]['perturbation'] = test_dataset.perturbations[pert_idx]
        test_dataset._pert_encoding_standardized = True
    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True
    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names

def objective(trial):
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler, common_genes_info
    n_hidden = trial.suggest_categorical('n_hidden', [256, 512, 1024])
    n_layers = trial.suggest_int('n_layers', 1, 3)
    n_heads = trial.suggest_categorical('n_heads', [4, 8])
    dropout = trial.suggest_float('dropout', 0.1, 0.3)
    attention_dropout = trial.suggest_float('attention_dropout', 0.1, 0.2)
    ffn_dropout = trial.suggest_float('ffn_dropout', 0.1, 0.2)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-3, log=True)
    weight_decay = trial.suggest_float('weight_decay', 1e-5, 1e-4, log=True)
    batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])
    scaler = StandardScaler()
    train_data = train_adata.X.toarray() if scipy.sparse.issparse(train_adata.X) else train_adata.X
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0
    train_dataset = ControlPerturbedDataset(
        train_adata, scaler=scaler, augment=True, is_train=True, common_genes_info=common_genes_info)
    test_perturbation_names = list(test_adata.obs['perturbation'].unique())
    pert_dim, _ = standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=test_perturbation_names)
    n_genes = train_dataset.n_genes
    print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
    model = GALAModel(
        input_dim=n_genes,
        output_dim=n_genes,
        pert_dim=pert_dim,
        hidden_dim=n_hidden,
        n_layers=n_layers,
        n_heads=n_heads,
        dropout=dropout,
        attention_dropout=attention_dropout,
        ffn_dropout=ffn_dropout
    ).to(device)
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=learning_rate,
        weight_decay=weight_decay,
        betas=(0.9, 0.999)
    )
    train_size = int(0.8 * len(train_dataset))
    val_size = len(train_dataset) - train_size
    train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(
        optimizer,
        max_lr=learning_rate,
        epochs=3,
        steps_per_epoch=len(train_loader),
        pct_start=0.1,
        anneal_strategy='cos'
    )
```

```
    best_val_loss = float('inf')
    patience = 3
    patience_counter = 0
    for epoch in tqdm(range(3), desc="Hyperparameter Trial"):
        train_loss = train_model(model, train_loader, optimizer, scheduler, device, max_pert_dim=pert_dim)
        val_metrics = evaluate_model(model, val_loader, device, max_pert_dim=pert_dim)
        val_loss = val_metrics['loss']
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break
    return best_val_loss

def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler, common_genes_info
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print_log(f'Training started at: {timestamp}')
    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print_log(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print_log(f'Warning: Specified GPU {gpu_id} does not exist, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print_log(f'Using GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print_log(f'Using GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print_log('CUDA not available, using CPU')
    print_log('Loading data...')
    train_path = "/datasets/AdamsonWeissman2016_train_processed.h5ad"
    test_path = "/datasets/AdamsonWeissman2016_test_processed.h5ad"
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(f"Data files not found: {train_path} or {test_path}")
    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print_log(f'Training data shape: {train_adata.shape}')
    print_log(f'Test data shape: {test_adata.shape}')
    print_log("Processing gene set consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print_log(f"Training genes: {len(train_genes)}")
    print_log(f"Test genes: {len(test_genes)}")
    print_log(f"Common genes: {len(common_genes)}")
    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(gene) for gene in common_genes]
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    scaler = StandardScaler()
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0
    common_genes_info = {
        'genes': common_genes,
        'train_idx': train_gene_idx,
        'test_idx': test_gene_idx
    }
    train_dataset = ControlPerturbedDataset(
        train_adata,
        perturbation_key='perturbation',
        scaler=scaler,
        augment=True,
        is_train=True,
        common_genes_info=common_genes_info
    )
    train_control_baseline = None
    if hasattr(train_dataset, 'control_baseline') and train_dataset.control_baseline is not None:
        train_control_baseline = train_dataset.control_baseline
        print_log(f"Extracted training set control baseline (shape: {train_control_baseline.shape}) for test set")
    test_dataset = ControlPerturbedDataset(
        test_adata,
        perturbation_key='perturbation',
        scaler=scaler,
        augment=False,
        is_train=False,
        common_genes_info=common_genes_info,
        train_control_baseline=train_control_baseline
    )
    optuna.logging.set_verbosity(optuna.logging.WARNING)
    print_log("Starting hyperparameter optimization...")
    study = optuna.create_study(
        direction='minimize',
        sampler=optuna.samplers.TPESampler(seed=42)
    )
    study.optimize(objective, n_trials=50, show_progress_bar=True)
    best_params = study.best_params
    print_log("Training final model with best parameters...")
    train_dataset = ControlPerturbedDataset(
        train_adata,
        perturbation_key='perturbation',
        scaler=scaler,
```

```
    augment=True,
    is_train=True,
    common_genes_info=common_genes_info
)
train_control_baseline = None
if hasattr(train_dataset, 'control_baseline') and train_dataset.control_baseline is not None:
    train_control_baseline = train_dataset.control_baseline
    print_log(f"Extracted training set control baseline (shape: {train_control_baseline.shape}) for test set")
test_dataset = ControlPerturbedDataset(
    test_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info,
    train_control_baseline=train_control_baseline
)
pert_dim, all_pert_names = standardize_perturbation_encoding(train_dataset, test_dataset)
n_genes = train_dataset.n_genes
print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
model = GALAModel(
    input_dim=n_genes,
    output_dim=n_genes,
    pert_dim=pert_dim,
    hidden_dim=best_params['n_hidden'],
    n_layers=best_params['n_layers'],
    n_heads=best_params['n_heads'],
    dropout=best_params['dropout'],
    attention_dropout=best_params['attention_dropout'],
    ffn_dropout=best_params['ffn_dropout']
).to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=best_params['learning_rate'],
    weight_decay=best_params['weight_decay'],
    betas=(0.9, 0.999)
)
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
train_loader = DataLoader(train_subset, batch_size=best_params['batch_size'], shuffle=True)
val_loader = DataLoader(val_subset, batch_size=best_params['batch_size'], shuffle=False)
scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=best_params['learning_rate'],
    epochs=200,
    steps_per_epoch=len(train_loader),
    pct_start=0.1,
    anneal_strategy='cos'
)
test_loader = DataLoader(test_dataset, batch_size=best_params['batch_size'], shuffle=False)
print_log("Starting training...")
best_loss = float('inf')
best_model = None
best_epoch = 0
patience = 20
patience_counter = 0
for epoch in tqdm(range(200), desc="Training Epochs"):
    train_loss = train_model(model, train_loader, optimizer, scheduler, device,
                             max_pert_dim=pert_dim)
    val_metrics = evaluate_model(model, val_loader, device,
                                 max_pert_dim=pert_dim)
    print_log(f'Epoch {epoch+1}/200:')
    print_log(f'Train Loss: {train_loss:.4f}')
    print_log(f'Validation Loss: {val_metrics["loss"]:.4f}')
    if val_metrics["loss"] < best_loss:
        best_loss = val_metrics["loss"]
        best_epoch = epoch
        patience_counter = 0
        best_model = model.state_dict()
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'loss': best_loss,
            'metrics': val_metrics,
            'best_params': best_params,
            'scaler': scaler,
            'common_genes_info': common_genes_info,
            'perturbation_encoding': {
                'pert_dim': pert_dim,
                'all_pert_names': all_pert_names
            }
        }, f'adamson_weissman_best_model_{timestamp}.pt')
        print_log(f"Saved best model with validation loss: {best_loss:.4f}")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print_log(f'Early stopping at epoch {epoch+1}')
            break
model.load_state_dict(best_model)
print_log('Evaluating final model on test set...')
test_metrics = evaluate_model(model, test_loader, device,
                              max_pert_dim=pert_dim, test_dataset=test_dataset)
best_params_str = str(best_params)
if len(best_params_str) > 50:
    best_params_str = best_params_str[:50] + '...'
results_df = pd.DataFrame({
    'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
    'Value': [
        test_metrics["MSE"],
        test_metrics["PCC"],
        test_metrics["R2"],
        test_metrics["MSE_DE"],
```

```
            test_metrics["PCC_DE"],
            test_metrics["R2_DE"]
        ],
        'Best_Params': [best_params_str] * 6
    })
    results_df.to_csv(f'adamson_weissman_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(index=False, float_format=lambda x: '{:.6f}'.format(x)))
    torch.save({
        'epoch': best_epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'scheduler_state_dict': scheduler.state_dict(),
        'loss': test_metrics["loss"],
        'metrics': test_metrics,
        'best_params': best_params,
        'evaluation_results': test_metrics,
        'scaler': scaler,
        'common_genes_info': common_genes_info,
        'perturbation_encoding': {
            'pert_dim': pert_dim,
            'all_pert_names': all_pert_names
        },
        'model_config': {
            'input_dim': n_genes,
            'output_dim': n_genes,
            'pert_dim': pert_dim,
            'hidden_dim': best_params['n_hidden'],
            'n_layers': best_params['n_layers'],
            'n_heads': best_params['n_heads'],
            'dropout': best_params['dropout'],
            'attention_dropout': best_params['attention_dropout'],
            'ffn_dropout': best_params['ffn_dropout']
        }
    }, f'adamson_weissman_final_model_{timestamp}.pt')
    print_log(f"Final model with evaluation results saved to: adamson_weissman_final_model_{timestamp}.pt")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='GALA-Former Model: AdamsonWeissman2016')
    parser.add_argument('--gpu', type=int, default=None, help='GPU ID to use')
    args = parser.parse_args()
    print("=" * 60)
    print("GALA-Former Model Training for AdamsonWeissman2016")
    print("=" * 60)
    if args.gpu is not None:
        print(f"Using GPU: {args.gpu}")
    else:
        print("Using default GPU settings")
    main(gpu_id=args.gpu)
```

D.2 *scGen-X* for Norman CRISPRa (scRNA-seq)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
import time
from datetime import datetime
import argparse
import warnings
from typing import Dict, List, Tuple, Optional, Union
import random
warnings.filterwarnings('ignore')
def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")

def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
scaler = None
common_genes_info = None
class ControlPerturbedDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', scaler=None,
                 augment=False, is_train=True, common_genes_info=None, train_control_baseline=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.augment = augment
        self.is_train = is_train
        self.common_genes_info = common_genes_info
        self.train_control_baseline = train_control_baseline
        if common_genes_info is not None:
```

```
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X
        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)
        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)
        if not is_log1p_already:
            data = np.log1p(data)
        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            if not is_standardized_already:
                data = self.scaler.transform(data)
        data = np.clip(data, -10, 10)
        data = data / 10.0
        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self._create_control_perturbed_pairs()
        print_log(f"Dataset created: {len(self.pairs)} pairs, {self.expression_data.shape[1]} features")
    def _create_control_perturbed_pairs(self):
        perturbation_col = self.adata.obs[self.perturbation_key].astype(str)
        negctrl_mask = perturbation_col.str.contains('NegCtrl', case=False, na=False)
        lower_labels = perturbation_col.str.lower()
        exact_control_mask = (lower_labels == 'control')
        other_control_mask = perturbation_col.isin(['Control', 'ctrl', 'Ctrl', 'CONTROL'])
        contains_control_mask = perturbation_col.str.contains('control|ctrl', case=False, na=False)
        control_mask = exact_control_mask | negctrl_mask | other_control_mask | contains_control_mask
        control_indices = np.where(control_mask)[0]
        perturbed_indices = np.where(~control_mask)[0]
        print_log(f"Control samples: {len(control_indices)}")
        print_log(f"Perturbed samples: {len(perturbed_indices)}")
        if len(control_indices) == 0:
            unique_perturbations = perturbation_col.unique()[:10]
            print_log(f"DEBUG: Sample perturbation values: {unique_perturbations}")
            print_log("DEBUG: Attempting to find control samples with alternative patterns...")
        self.perturbation_names = list(perturbation_col.unique())
        self.perturbations = pd.get_dummies(perturbation_col).values.astype(np.float32)
        print_log(f"Perturbation dimension: {self.perturbations.shape[1]}")
        if len(control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
                    "Please ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                if self.train_control_baseline is None:
                    raise ValueError(
                        "CRITICAL ERROR: Test set has no control samples AND no training control baseline provided! "
                        "This is required for unseen perturbation prediction. "
                        "Please ensure training set has control samples."
                    )
                else:
                    print_log("=" * 60)
                    print_log("Test set: No control samples found in test data.")
                    print_log("Using training set control baseline (CORRECT for unseen perturbation task).")
                    print_log("=" * 60)
                    self.avg_baseline = self.train_control_baseline
                    self.use_avg_baseline = True
        else:
            print_log(f"Using {len(control_indices)} true control samples as baseline")
            control_baseline = np.mean(self.expression_data[control_indices], axis=0)
            self.control_baseline = control_baseline
            self.use_avg_baseline = False
            self.avg_baseline = None
        self.pairs = []
        for i, perturbed_idx in enumerate(perturbed_indices):
            if self.use_avg_baseline:
                baseline_idx = -1
            else:
                available_controls = control_indices[control_indices != perturbed_idx]
                if len(available_controls) > 0:
                    if self.is_train:
                        baseline_idx = np.random.choice(available_controls)
                    else:
                        baseline_idx = available_controls[i % len(available_controls)]
                else:
                    if self.is_train:
                        baseline_idx = np.random.choice(control_indices)
                    else:
```

```
                    baseline_idx = control_indices[i % len(control_indices)]
                self.pairs.append({
                    'baseline_idx': baseline_idx,
                    'perturbed_idx': perturbed_idx,
                    'perturbation': self.perturbations[perturbed_idx]
                })
        if len(control_indices) > 0 and len(perturbed_indices) > 0:
            for i, control_idx in enumerate(control_indices):
                if self.is_train:
                    target_idx = np.random.choice(perturbed_indices)
                else:
                    target_idx = perturbed_indices[i % len(perturbed_indices)]
                self.pairs.append({
                    'baseline_idx': control_idx,
                    'perturbed_idx': target_idx,
                    'perturbation': self.perturbations[target_idx]
                })
        elif len(control_indices) > 1:
            for i, control_idx in enumerate(control_indices):
                other_controls = control_indices[control_indices != control_idx]
                if len(other_controls) > 0:
                    if self.is_train:
                        baseline_idx = np.random.choice(other_controls)
                    else:
                        baseline_idx = other_controls[i % len(other_controls)]
                    self.pairs.append({
                        'baseline_idx': baseline_idx,
                        'perturbed_idx': control_idx,
                        'perturbation': self.perturbations[control_idx]
                    })
        print_log(f"Total pairs created: {len(self.pairs)}")
    def __len__(self):
        return len(self.pairs)
    def __getitem__(self, idx):
        pair = self.pairs[idx]
        if pair['baseline_idx'] == -1:
            x_control = self.avg_baseline.copy()
        else:
            x_control = self.expression_data[pair['baseline_idx']]
        x_perturbed = self.expression_data[pair['perturbed_idx']]
        pert = pair['perturbation']
        if pair['baseline_idx'] != -1 and pair['baseline_idx'] == pair['perturbed_idx']:
            print_log(f"WARNING: Found self-pair (baseline_idx == perturbed_idx). This should not happen!")
            if hasattr(self, 'avg_baseline') and self.avg_baseline is not None:
                x_control = self.avg_baseline.copy()
            else:
                x_control = self.expression_data[0].copy()
            x_perturbed = self.expression_data[pair['perturbed_idx']]
            pert = pair['perturbation']
        x_perturbed_original = self.original_expression_data[pair['perturbed_idx']]
        x_target_abs = x_perturbed_original
        if self.augment and self.is_train:
            noise = np.random.normal(0, 0.05, x_control.shape)
            x_control = x_control + noise
            mask = np.random.random(x_control.shape) > 0.05
            x_control = x_control * mask
        return (torch.FloatTensor(x_control),
                torch.FloatTensor(pert),
                torch.FloatTensor(x_target_abs))
    def get_pert_dim(self):
        return self.perturbations.shape[1]
class VAEEncoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)
    def forward(self, x):
        h = F.gelu(self.fc1(x))
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        return z, mu, logvar
class VAEDecoder(nn.Module):
    def __init__(self, latent_dim, output_dim, hidden_dim):
        super().__init__()
        self.fc1 = nn.Linear(latent_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
    def forward(self, z):
        h = F.gelu(self.fc1(z))
        return self.fc2(h)
class PerturbationEmbedding(nn.Module):
    def __init__(self, pert_dim, emb_dim):
        super().__init__()
        self.embedding = nn.Linear(pert_dim, emb_dim)
    def forward(self, pert):
        return self.embedding(pert)
class SingleTokenAttention(nn.Module):
    def __init__(self, query_dim, key_dim, value_dim, hidden_dim, n_heads=8, dropout=0.1):
        super().__init__()
        assert hidden_dim % n_heads == 0, "hidden_dim must be divisible by n_heads"
        self.n_heads = n_heads
        self.head_dim = hidden_dim // n_heads
        self.hidden_dim = hidden_dim
        self.query_proj = nn.Linear(query_dim, hidden_dim)
        self.key_proj = nn.Linear(key_dim, hidden_dim)
        self.value_proj = nn.Linear(value_dim, hidden_dim)
        self.output_proj = nn.Linear(hidden_dim, hidden_dim)
        self.dropout = nn.Dropout(dropout)
        self.scale = self.head_dim ** -0.5
    def forward(self, query, key, value):
        batch_size = query.size(0)
```

```
        Q = self.query_proj(query).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        K = self.key_proj(key).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        V = self.value_proj(value).view(batch_size, 1, self.n_heads, self.head_dim).transpose(1, 2)
        scores = torch.matmul(Q, K.transpose(-2, -1)) * self.scale
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)
        attended = torch.matmul(attn_weights, V)
        attended = attended.transpose(1, 2).contiguous().view(batch_size, self.hidden_dim)
        output = self.output_proj(attended)
        return output
class HybridAttentionModel(nn.Module):
    def __init__(self, input_dim, output_dim, pert_dim, hidden_dim=512, n_layers=2, n_heads=8,
                 dropout=0.1, attention_dropout=0.1, ffn_dropout=0.1, activation='gelu',
                 use_vae=False, vae_latent_dim=64, vae_hidden_dim=256,
                 use_pert_emb=False, pert_emb_dim=32, vae_beta=1.0,
                 model_in_dim=128, bottleneck_dims=(2048, 512), use_bottleneck=True):
        super(HybridAttentionModel, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.model_in_dim = model_in_dim
        self.pert_dim = pert_dim
        self.hidden_dim = hidden_dim
        self.n_heads = n_heads
        self.use_bottleneck = use_bottleneck
        self.use_vae = use_vae
        self.vae_beta = vae_beta
        self.use_pert_emb = use_pert_emb
        if self.use_bottleneck:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, bottleneck_dims[0]),
                nn.LayerNorm(bottleneck_dims[0]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[0], bottleneck_dims[1]),
                nn.LayerNorm(bottleneck_dims[1]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[1], self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
            )
        else:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
                nn.GELU()
            )
        if use_vae:
            self.vae_encoder = VAEEncoder(self.model_in_dim, vae_latent_dim, vae_hidden_dim)
            self.vae_decoder = VAEDecoder(vae_latent_dim, self.model_in_dim, vae_hidden_dim)
            expr_out_dim = vae_latent_dim
        else:
            self.expression_encoder = nn.Sequential(
                nn.Linear(self.model_in_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout),
                nn.Linear(hidden_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout),
                nn.Linear(hidden_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout)
            )
            expr_out_dim = hidden_dim
        if use_pert_emb:
            self.pert_encoder = PerturbationEmbedding(pert_dim, pert_emb_dim)
            pert_out_dim = pert_emb_dim
        else:
            self.pert_encoder = nn.Sequential(
                nn.Linear(pert_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout)
            )
            pert_out_dim = hidden_dim
        self.cross_attention = SingleTokenAttention(
            query_dim=pert_out_dim,
            key_dim=expr_out_dim,
            value_dim=expr_out_dim,
            hidden_dim=hidden_dim,
            n_heads=n_heads,
            dropout=attention_dropout
        )
        mlp_layers = []
        for _ in range(n_layers):
            mlp_layers.extend([
                nn.Linear(hidden_dim, hidden_dim * 2),
                nn.LayerNorm(hidden_dim * 2),
                nn.GELU() if activation == 'gelu' else nn.ReLU(),
                nn.Dropout(ffn_dropout),
                nn.Linear(hidden_dim * 2, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU() if activation == 'gelu' else nn.ReLU(),
                nn.Dropout(ffn_dropout)
            ])
        self.mlp = nn.Sequential(*mlp_layers) if mlp_layers else nn.Identity()
        self.fusion = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.output = nn.Sequential(
```

```
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, output_dim)
        )
        self.apply(self._init_weights)
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)
    def forward(self, x_control, pert):
        assert x_control.dim() == 2 and x_control.size(1) == self.input_dim, \
            f"Expected x_control shape [B, {self.input_dim}], got {x_control.shape}"
        x_control_proj = self.input_proj(x_control)
        vae_kl = 0
        vae_recon = None
        if self.use_vae:
            z, mu, logvar = self.vae_encoder(x_control_proj)
            vae_recon = self.vae_decoder(z)
            vae_kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1).mean()
            expr_feat = z
        else:
            expr_feat = self.expression_encoder(x_control_proj)
        pert_feat = self.pert_encoder(pert)
        attended_feat = self.cross_attention(pert_feat, expr_feat, expr_feat)
        x_trans = self.mlp(attended_feat)
        fused = self.fusion(x_trans)
        # [OK] Changed from delta-based to absolute-based: directly predict absolute target value
        output = self.output(fused)
        return output, vae_recon, vae_kl, x_control_proj
def train_model(model, train_loader, optimizer, scheduler, device, vae_beta=1.0, max_pert_dim=None):
    model.train()
    total_loss = 0
    for batch in tqdm(train_loader, desc="Training"):
        x_control, pert, x_target_abs = batch
        x_control, pert, x_target_abs = x_control.to(device), pert.to(device), x_target_abs.to(device)
        if max_pert_dim is not None and pert.shape[1] < max_pert_dim:
            pad_size = max_pert_dim - pert.shape[1]
            pert = F.pad(pert, (0, pad_size), mode='constant', value=0)
        optimizer.zero_grad()
        output, vae_recon, vae_kl, x_control_proj = model(x_control, pert)
        main_loss = F.mse_loss(output, x_target_abs)
        vae_loss = 0
        if vae_recon is not None:
            vae_loss = F.mse_loss(vae_recon, x_control_proj)
        loss = main_loss + vae_beta * vae_kl + 0.1 * vae_loss
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        scheduler.step()
        total_loss += loss.item()
    return total_loss / len(train_loader)
def calculate_detailed_metrics(pred, true, control_baseline=None, perturbations=None):
    n_samples, n_genes = true.shape
    mse = np.mean((pred - true) ** 2)
    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0
    true_mean_vector = np.mean(true, axis=0)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_vector) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0

    if control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) / (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
                    pred_de_flat = pred_de.flatten()
```

```
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_vector = np.mean(true_de, axis=0)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                mse_de = np.mean((pred_de - true_de) ** 2)
                pred_de_flat = pred_de.flatten()
                true_de_flat = true_de.flatten()
                if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                    pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0
                true_de_mean_vector = np.mean(true_de, axis=0)
                ss_res_de = np.sum((true_de - pred_de) ** 2)
                ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                if ss_tot_de > 1e-10:
                    r2_de = 1.0 - (ss_res_de / ss_tot_de)
                else:
                    r2_de = 0.0
                if np.isnan(r2_de):
                    r2_de = 0.0
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan

    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }

def evaluate_model(model, test_loader, device, vae_beta=1.0, max_pert_dim=None, test_dataset=None):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    all_controls = []
    all_perts = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            x_control, pert, x_target_abs = batch
            x_control, pert, x_target_abs = x_control.to(device), pert.to(device), x_target_abs.to(device)
            if max_pert_dim is not None and pert.shape[1] < max_pert_dim:
                pad_size = max_pert_dim - pert.shape[1]
                pert = F.pad(pert, (0, pad_size), mode='constant', value=0)
            output, vae_recon, vae_kl, x_control_proj = model(x_control, pert)
            main_loss = F.mse_loss(output, x_target_abs)
            vae_loss = 0
            if vae_recon is not None:
                vae_loss = F.mse_loss(vae_recon, x_control_proj)
            loss = main_loss + vae_beta * vae_kl + 0.1 * vae_loss
            total_loss += loss.item()
            all_targets.append(x_target_abs.cpu().numpy())
            all_predictions.append(output.cpu().numpy())
            all_controls.append(x_control.cpu().numpy())
            all_perts.append(pert.cpu().numpy())
    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_controls = np.concatenate(all_controls, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)


    r2 = r2_score(all_targets, all_predictions)
    if np.isnan(r2):
        r2 = 0.0
    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pearson = pearsonr(true_flat, pred_flat)[0]
```

```
        if np.isnan(pearson):
            pearson = 0.0
    else:
        pearson = 0.0


    if test_dataset is not None and hasattr(test_dataset, 'control_baseline') and test_dataset.control_baseline is not None and not (hasattr(test_dataset, 'use_avg_baseline') and
        test_dataset.use_avg_baseline):
        control_baseline = test_dataset.control_baseline.reshape(1, -1) if test_dataset.control_baseline.ndim == 1 else test_dataset.control_baseline
    elif test_dataset is not None and hasattr(test_dataset, 'use_avg_baseline') and test_dataset.use_avg_baseline:
        control_baseline = None
    else:
        unique_controls = np.unique(all_controls, axis=0)
        if len(unique_controls) > 0:
            control_baseline = all_controls if len(all_controls) > 0 else None
        else:
            control_baseline = None
    detailed_metrics = calculate_detailed_metrics(all_predictions, all_targets, control_baseline=control_baseline, perturbations=all_perts)

    return {
        'loss': total_loss / len(test_loader),
        'r2': r2,
        'pearson': pearson,
        'MSE': detailed_metrics['MSE'],
        'PCC': detailed_metrics['PCC'],
        'R2': detailed_metrics['R2'],
        'MSE_DE': detailed_metrics['MSE_DE'],
        'PCC_DE': detailed_metrics['PCC_DE'],
        'R2_DE': detailed_metrics['R2_DE']
    }
def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(list(set(train_dataset.perturbation_names + test_dataset.perturbation_names)))
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []
    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))
    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(
        columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)
    for i, pair in enumerate(train_dataset.pairs):
        pert_idx = pair['perturbed_idx']
        train_dataset.pairs[i]['perturbation'] = train_dataset.perturbations[pert_idx]
    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(np.float32)
        for i, pair in enumerate(test_dataset.pairs):
            pert_idx = pair['perturbed_idx']
            test_dataset.pairs[i]['perturbation'] = test_dataset.perturbations[pert_idx]
        test_dataset._pert_encoding_standardized = True
    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True
    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names
def objective(trial):
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler, common_genes_info
    n_hidden = trial.suggest_categorical('n_hidden', [256, 512, 1024])
    n_layers = trial.suggest_int('n_layers', 1, 3)
    n_heads = trial.suggest_categorical('n_heads', [4, 8])
    dropout = trial.suggest_float('dropout', 0.1, 0.3)
    attention_dropout = trial.suggest_float('attention_dropout', 0.1, 0.2)
    ffn_dropout = trial.suggest_float('ffn_dropout', 0.1, 0.2)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-3, log=True)
    weight_decay = trial.suggest_float('weight_decay', 1e-5, 1e-4, log=True)
    batch_size = trial.suggest_categorical('batch_size', [32, 64, 128])
    use_vae = trial.suggest_categorical('use_vae', [True, False])
    vae_latent_dim = trial.suggest_categorical('vae_latent_dim', [32, 64])
    vae_hidden_dim = trial.suggest_categorical('vae_hidden_dim', [128, 256])
    use_pert_emb = trial.suggest_categorical('use_pert_emb', [True, False])
    pert_emb_dim = trial.suggest_categorical('pert_emb_dim', [16, 32])
    vae_beta = trial.suggest_float('vae_beta', 0.1, 0.5)
    scaler = StandardScaler()
    train_data = train_adata.X.toarray() if scipy.sparse.issparse(train_adata.X) else train_adata.X
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0
    train_dataset = ControlPerturbedDataset(
```

```
        train_adata, scaler=scaler, augment=True, is_train=True, common_genes_info=common_genes_info)
    test_perturbation_names = list(test_adata.obs['perturbation'].unique())
    pert_dim, _ = standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=test_perturbation_names)
    n_genes = train_dataset.n_genes
    print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
    model = HybridAttentionModel(
        input_dim=n_genes,
        output_dim=n_genes,
        pert_dim=pert_dim,
        hidden_dim=n_hidden,
        n_layers=n_layers,
        n_heads=n_heads,
        dropout=dropout,
        attention_dropout=attention_dropout,
        ffn_dropout=ffn_dropout,
        use_vae=use_vae,
        vae_latent_dim=vae_latent_dim,
        vae_hidden_dim=vae_hidden_dim,
        use_pert_emb=use_pert_emb,
        pert_emb_dim=pert_emb_dim,
        vae_beta=vae_beta
    ).to(device)
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=learning_rate,
        weight_decay=weight_decay,
        betas=(0.9, 0.999)
    )
    train_size = int(0.8 * len(train_dataset))
    val_size = len(train_dataset) - train_size
    train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(
        optimizer,
        max_lr=learning_rate,
        epochs=3,
        steps_per_epoch=len(train_loader),
        pct_start=0.1,
        anneal_strategy='cos'
    )
    best_val_loss = float('inf')
    patience = 3
    patience_counter = 0
    for epoch in tqdm(range(3), desc="Hyperparameter Trial"):
        train_loss = train_model(model, train_loader, optimizer, scheduler, device, vae_beta=vae_beta, max_pert_dim=pert_dim)
        val_metrics = evaluate_model(model, val_loader, device, vae_beta=vae_beta, max_pert_dim=pert_dim)
        val_loss = val_metrics['loss']
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break
    return best_val_loss
def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler, common_genes_info
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print_log(f'Training started at: {timestamp}')
    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print_log(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print_log(f'Warning: Specified GPU {gpu_id} does not exist, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print_log(f'Using GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print_log(f'Using GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print_log('CUDA not available, using CPU')
    print_log('Loading data...')
    train_path = "/datasets/NormanWeissman2019_filtered_train_processed_unseenpert1.h5ad"
    test_path = "/datasets/NormanWeissman2019_filtered_test_processed_unseenpert1.h5ad"
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(f"Data files not found: {train_path} or {test_path}")
    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print_log(f'Training data shape: {train_adata.shape}')
    print_log(f'Test data shape: {test_adata.shape}')
    print_log("Processing gene set consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print_log(f"Training genes: {len(train_genes)}")
    print_log(f"Test genes: {len(test_genes)}")
    print_log(f"Common genes: {len(common_genes)}")
    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(gene) for gene in common_genes]
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    scaler = StandardScaler()
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
```

```
train_data = train_data / 10.0
common_genes_info = {
    'genes': common_genes,
    'train_idx': train_gene_idx,
    'test_idx': test_gene_idx
}
train_dataset = ControlPerturbedDataset(
    train_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=True,
    is_train=True,
    common_genes_info=common_genes_info
)
train_control_baseline = None
if hasattr(train_dataset, 'control_baseline') and train_dataset.control_baseline is not None:
    train_control_baseline = train_dataset.control_baseline
    print_log(f"Extracted training set control baseline (shape: {train_control_baseline.shape}) for test set")
test_dataset = ControlPerturbedDataset(
    test_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info,
    train_control_baseline=train_control_baseline
)

optuna.logging.set_verbosity(optuna.logging.WARNING)
print_log("Starting hyperparameter optimization...")
study = optuna.create_study(
    direction='minimize',
    sampler=optuna.samplers.TPESampler(seed=42)
)
study.optimize(objective, n_trials=50, show_progress_bar=True)
best_params = study.best_params
print_log("Training final model with best parameters...")
train_dataset = ControlPerturbedDataset(
    train_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=True,
    is_train=True,
    common_genes_info=common_genes_info
)
train_control_baseline = None
if hasattr(train_dataset, 'control_baseline') and train_dataset.control_baseline is not None:
    train_control_baseline = train_dataset.control_baseline
    print_log(f"Extracted training set control baseline (shape: {train_control_baseline.shape}) for test set")
test_dataset = ControlPerturbedDataset(
    test_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info,
    train_control_baseline=train_control_baseline
)
pert_dim, all_pert_names = standardize_perturbation_encoding(train_dataset, test_dataset)
n_genes = train_dataset.n_genes
print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
model = HybridAttentionModel(
    input_dim=n_genes,
    output_dim=n_genes,
    pert_dim=pert_dim,
    hidden_dim=best_params['n_hidden'],
    n_layers=best_params['n_layers'],
    n_heads=best_params['n_heads'],
    dropout=best_params['dropout'],
    attention_dropout=best_params['attention_dropout'],
    ffn_dropout=best_params['ffn_dropout'],
    use_vae=best_params['use_vae'],
    vae_latent_dim=best_params['vae_latent_dim'],
    vae_hidden_dim=best_params['vae_hidden_dim'],
    use_pert_emb=best_params['use_pert_emb'],
    pert_emb_dim=best_params['pert_emb_dim'],
    vae_beta=best_params['vae_beta']
).to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=best_params['learning_rate'],
    weight_decay=best_params['weight_decay'],
    betas=(0.9, 0.999)
)
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
train_loader = DataLoader(train_subset, batch_size=best_params['batch_size'], shuffle=True)
val_loader = DataLoader(val_subset, batch_size=best_params['batch_size'], shuffle=False)
scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,
    max_lr=best_params['learning_rate'],
    epochs=200,
    steps_per_epoch=len(train_loader),
    pct_start=0.1,
    anneal_strategy='cos'
)
test_loader = DataLoader(test_dataset, batch_size=best_params['batch_size'], shuffle=False)
print_log("Starting training...")
best_loss = float('inf')
best_model = None
best_epoch = 0
patience = 20
patience_counter = 0
for epoch in tqdm(range(200), desc="Training Epochs"):
```

```
        train_loss = train_model(model, train_loader, optimizer, scheduler, device,
                                 vae_beta=best_params['vae_beta'], max_pert_dim=pert_dim)
        val_metrics = evaluate_model(model, val_loader, device,
                                     vae_beta=best_params['vae_beta'], max_pert_dim=pert_dim)
        print_log(f'Epoch {epoch+1}/200:')
        print_log(f'Train Loss: {train_loss:.4f}')
        print_log(f'Validation Loss: {val_metrics["loss"]:.4f}')
        if val_metrics["loss"] < best_loss:
            best_loss = val_metrics["loss"]
            best_epoch = epoch
            patience_counter = 0
            best_model = model.state_dict()
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss': best_loss,
                'metrics': val_metrics,
                'best_params': best_params,
                'scaler': scaler,
                'common_genes_info': common_genes_info,
                'perturbation_encoding': {
                    'pert_dim': pert_dim,
                    'all_pert_names': all_pert_names
                }
            }, f'model_best_{timestamp}.pt')
            print_log(f"Saved best model with validation loss: {best_loss:.4f}")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print_log(f'Early stopping at epoch {epoch+1}')
                break
    model.load_state_dict(best_model)
    print_log('Evaluating final model on test set...')
    test_metrics = evaluate_model(model, test_loader, device,
                                  vae_beta=best_params['vae_beta'], max_pert_dim=pert_dim, test_dataset=test_dataset)



    best_params_str = str(best_params)
    if len(best_params_str) > 50:
        best_params_str = best_params_str[:50] + '...'
    results_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [
            test_metrics["MSE"],
            test_metrics["PCC"],
            test_metrics["R2"],
            test_metrics["MSE_DE"],
            test_metrics["PCC_DE"],
            test_metrics["R2_DE"]
        ],
        'Best_Params': [best_params_str] * 6
    })
    results_df.to_csv(f'norman_weissman_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(index=False, float_format=lambda x: '{:.6f}'.format(x)))


    torch.save({
        'epoch': best_epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'scheduler_state_dict': scheduler.state_dict(),
        'loss': test_metrics["loss"],
        'metrics': test_metrics,
        'best_params': best_params,
        'evaluation_results': test_metrics,
        'scaler': scaler,
        'common_genes_info': common_genes_info,
        'perturbation_encoding': {
            'pert_dim': pert_dim,
            'all_pert_names': all_pert_names
        },
        'model_config': {
            'input_dim': n_genes,
            'output_dim': n_genes,
            'pert_dim': pert_dim,
            'hidden_dim': best_params['n_hidden'],
            'n_layers': best_params['n_layers'],
            'n_heads': best_params['n_heads'],
            'dropout': best_params['dropout'],
            'attention_dropout': best_params['attention_dropout'],
            'ffn_dropout': best_params['ffn_dropout'],
            'use_vae': best_params['use_vae'],
            'vae_latent_dim': best_params['vae_latent_dim'],
            'vae_hidden_dim': best_params['vae_hidden_dim'],
            'use_pert_emb': best_params['use_pert_emb'],
            'pert_emb_dim': best_params['pert_emb_dim'],
            'vae_beta': best_params['vae_beta']
        }
    }, f'model_final_{timestamp}.pt')
    print_log(f"Final model with evaluation results saved to: model_final_{timestamp}.pt")
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Model: Control -> Perturbed Prediction')
    parser.add_argument('--gpu', type=int, default=None, help='GPU ID to use')
    args = parser.parse_args()
    print("=" * 60)
    print("Hybrid Attention Model Training")
    print("=" * 60)
    if args.gpu is not None:
        print(f"Using GPU: {args.gpu}")
    else:
        print("Using default GPU settings")
```

```
    results_df = main(gpu_id=args.gpu)
```

D.3 *ChemCellFlow* for Srivatsan drug perturbation (scRNA-seq)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
import time
from datetime import datetime
import argparse
import warnings
from typing import Dict, List, Tuple, Optional, Union
import random
warnings.filterwarnings('ignore')
def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")

def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
scaler = None
class GeneExpressionDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', dose_key='dose_value',
                 scaler=None, augment=False, is_train=True, common_genes_info=None,
                 reference_perturbation=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.dose_key = dose_key
        self.augment = augment
        self.training = True
        self.is_train = is_train
        self.common_genes_info = common_genes_info
        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X
        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)
        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)
        if not is_log1p_already:
            data = np.log1p(data)
        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            if not is_standardized_already:
                data = self.scaler.transform(data)
        data = np.clip(data, -10, 10)
        data = data / 10.0
        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self.perturbations = pd.get_dummies(
            adata.obs[perturbation_key]).values.astype(np.float32)
        print_log(
            f"{'Training' if is_train else 'Test'} set perturbation dim: {self.perturbations.shape[1]}")
        self.perturbation_names = list(adata.obs[perturbation_key].unique())
```

```
    perturbation_labels = adata.obs[perturbation_key].astype(str).values
    perturbation_labels = np.array(perturbation_labels, dtype='U')
    lower_labels = np.char.lower(perturbation_labels)
    negctrl_mask = np.char.find(lower_labels, 'negctrl') >= 0
    control_mask = (lower_labels == 'control') | \
        (lower_labels == 'ctrl') | negctrl_mask
    self.reference_perturbation = reference_perturbation
    if len(np.where(control_mask)[0]) == 0 and reference_perturbation is not None:
        ref_mask = perturbation_labels == reference_perturbation
        control_mask = control_mask | ref_mask
    self._control_indices = np.where(control_mask)[0]
    self._non_control_indices = np.where(~control_mask)[0]
    self._pert_to_indices = {}
    for pert_name in self.perturbation_names:
        pert_mask = perturbation_labels == pert_name
        pert_indices = np.where(pert_mask)[0]
        pert_indices = pert_indices[~np.isin(pert_indices, self._control_indices)]
        if len(pert_indices) > 0:
            self._pert_to_indices[pert_name] = pert_indices
    self._non_control_pert_names = [name for name in self.perturbation_names
                                    if name not in ['control', 'Control', 'ctrl', 'Ctrl']
                                    and 'negctrl' not in name.lower()]
    if reference_perturbation is not None and reference_perturbation in self._non_control_pert_names:
        self._non_control_pert_names = [name for name in self._non_control_pert_names
                                        if name != reference_perturbation]
    if len(self._control_indices) == 0:
        if self.is_train:
            raise ValueError(
                "CRITICAL ERROR: Training set has NO control samples! "
                "Control samples are REQUIRED for perturbation prediction. "
                "This model performs control -> perturbed prediction, NOT autoencoder. "
                "Options:\n"
                "  1. Ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl').\n"
                "  2. Use 'reference_perturbation' parameter to specify a reference perturbation as baseline.\n"
                "     Example: reference_perturbation='SAHA' (for Srivatsan dataset)"
            )
        else:
            raise ValueError(
                "CRITICAL ERROR: Test set has NO control samples! "
                "For unseen perturbation prediction, test set should use training set control baseline. "
                "Please ensure training set has control samples or use reference_perturbation parameter."
            )
    self.rng = np.random.RandomState(42)
    if not self.is_train:
        self._create_fixed_pairs_for_test()
    if dose_key in adata.obs.columns:
        dose_values = pd.to_numeric(adata.obs[dose_key], errors='coerce')
        dose_values = dose_values.fillna(0.0)
        if dose_values.max() > dose_values.min():
            self.dose_values = (dose_values - dose_values.min()) / \
                (dose_values.max() - dose_values.min())
        else:
            self.dose_values = np.zeros_like(dose_values)
    else:
        self.dose_values = np.zeros(len(adata))
    print_log(
        f"Dataset created: {len(self.adata)} cells, {self.expression_data.shape[1]} features")
def _create_fixed_pairs_for_test(self):
    n = len(self.adata)
    self.pairs = [None] * n
    for idx in range(n):
        if idx in self._control_indices:
            baseline_idx = idx
            if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                pert_name = self._non_control_pert_names[idx % len(self._non_control_pert_names)]
                if pert_name in self._pert_to_indices and len(self._pert_to_indices[pert_name]) > 0:
                    pert_indices = self._pert_to_indices[pert_name]
                    target_idx = int(pert_indices[(idx // len(self._non_control_pert_names)) % len(pert_indices)])
                else:
                    target_idx = int(self._non_control_indices[idx % len(self._non_control_indices)])
            else:
                alternative_controls = self._control_indices[self._control_indices != idx]
                if len(alternative_controls) > 0:
                    target_idx = int(alternative_controls[idx % len(alternative_controls)])
                else:
                    target_idx = idx
        else:
            if len(self._control_indices) > 0:
                baseline_idx = int(self._control_indices[idx % len(self._control_indices)])
            else:
                baseline_idx = idx
            target_idx = idx
        self.pairs[idx] = (baseline_idx, target_idx)
def __len__(self):
    return len(self.adata)
def __getitem__(self, idx):
    x_current = self.expression_data[idx]
    pert = self.perturbations[idx]
    dose = self.dose_values[idx]
    if self.is_train:
        if idx in self._control_indices:
            x_baseline = x_current
            if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                pert_name = self.rng.choice(self._non_control_pert_names)
                if pert_name in self._pert_to_indices and len(self._pert_to_indices[pert_name]) > 0:
                    pert_indices = self._pert_to_indices[pert_name]
                    target_idx = int(self.rng.choice(pert_indices))
                    x_target = self.original_expression_data[target_idx]
                    pert_target = self.perturbations[target_idx]
                    dose_target = self.dose_values[target_idx]
                else:
                    target_idx = int(self.rng.choice(self._non_control_indices))
                    x_target = self.original_expression_data[target_idx]
                    pert_target = self.perturbations[target_idx]
                    dose_target = self.dose_values[target_idx]
```

```
                else:
                    alternative_controls = self._control_indices[self._control_indices != idx]
                    if len(alternative_controls) > 0:
                        target_idx = int(self.rng.choice(alternative_controls))
                        x_target = self.original_expression_data[target_idx]
                        pert_target = self.perturbations[target_idx]
                        dose_target = self.dose_values[target_idx]
                    else:
                        x_target = self.original_expression_data[idx]
                        pert_target = pert
                        dose_target = dose
            else:
                if len(self._control_indices) > 0:
                    baseline_idx = int(self.rng.choice(self._control_indices))
                    x_baseline = self.expression_data[baseline_idx]
                else:
                    x_baseline = x_current
                x_target = self.original_expression_data[idx]
                pert_target = pert
                dose_target = dose
        else:
            baseline_idx, target_idx = self.pairs[idx]
            if baseline_idx == idx and idx in self._control_indices:
                x_baseline = x_current
            else:
                x_baseline = self.expression_data[baseline_idx]
            x_target = self.original_expression_data[target_idx]
            if target_idx == idx:
                pert_target = pert
                dose_target = dose
            else:
                pert_target = self.perturbations[target_idx]
                dose_target = self.dose_values[target_idx]
        x_target_abs = x_target
        if self.augment and self.training:
            noise = np.random.normal(0, 0.05, x_baseline.shape)
            x_baseline = x_baseline + noise
            mask = np.random.random(x_baseline.shape) > 0.05
            x_baseline = x_baseline * mask
        x_baseline = x_baseline.astype(np.float32)
        pert_target = pert_target.astype(np.float32)
        dose_target = np.float32(dose_target)
        return torch.FloatTensor(x_baseline), torch.FloatTensor(pert_target), torch.FloatTensor([dose_target]), torch.FloatTensor(x_target_abs)
class GeneEncoder(nn.Module):
    def __init__(self, input_dim, latent_dim=128, hidden_dim=512, n_heads=8, n_layers=2, dropout=0.1):
        super(GeneEncoder, self).__init__()
        self.input_dim = input_dim
        self.latent_dim = latent_dim
        self.input_proj = nn.Linear(input_dim, hidden_dim)
        mlp_layers = []
        for _ in range(n_layers):
            mlp_layers.extend([
                nn.Linear(hidden_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout)
            ])
        self.mlp = nn.Sequential(*mlp_layers) if mlp_layers else nn.Identity()
        self.output_proj = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.LayerNorm(hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, latent_dim)
        )
    def forward(self, x):
        x = self.input_proj(x)
        x = self.mlp(x)
        z = self.output_proj(x)
        return z
class ChemEncoder(nn.Module):
    def __init__(self, pert_dim, dose_dim=1, chem_dim=64, hidden_dim=256, dropout=0.1):
        super(ChemEncoder, self).__init__()
        self.pert_dim = pert_dim
        self.dose_dim = dose_dim
        self.chem_dim = chem_dim
        self.pert_encoder = nn.Sequential(
            nn.Linear(pert_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.LayerNorm(hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.dose_encoder = nn.Sequential(
            nn.Linear(dose_dim, 32),
            nn.GELU(),
            nn.Linear(32, 32),
            nn.GELU()
        )
        self.fusion = nn.Sequential(
            nn.Linear(hidden_dim // 2 + 32, chem_dim),
            nn.LayerNorm(chem_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
    def forward(self, pert, dose):
        pert_feat = self.pert_encoder(pert)
        dose_feat = self.dose_encoder(dose)
        combined = torch.cat([pert_feat, dose_feat], dim=1)
        p = self.fusion(combined)
        return p
class ConditionalOT(nn.Module):
```

```
    def __init__(self, latent_dim, chem_dim, eps=0.1, max_iter=50):
        super(ConditionalOT, self).__init__()
        self.latent_dim = latent_dim
        self.chem_dim = chem_dim
        self.eps = eps
        self.max_iter = max_iter
    def sinkhorn(self, a, b, C, eps, max_iter):
        u = torch.zeros_like(a)
        v = torch.zeros_like(b)
        for _ in range(max_iter):
            u = eps * (torch.log(a + 1e-8) -
                       torch.logsumexp((u.unsqueeze(1) + v.unsqueeze(0) - C) / eps, dim=1))
            v = eps * (torch.log(b + 1e-8) -
                       torch.logsumexp((u.unsqueeze(1) + v.unsqueeze(0) - C) / eps, dim=0))
        P = torch.exp((u.unsqueeze(1) + v.unsqueeze(0) - C) / eps)
        return P
    def forward(self, z_control, z_perturbed, p):
        batch_size = z_control.size(0)
        a = torch.ones(batch_size, device=z_control.device) / batch_size
        b = torch.ones(batch_size, device=z_control.device) / batch_size
        C = torch.cdist(z_control, z_perturbed, p=2) ** 2
        P = self.sinkhorn(a, b, C, self.eps, self.max_iter)
        z_ot_pred = torch.mm(P, z_perturbed)
        return z_ot_pred, P
class FlowRefiner(nn.Module):
    def __init__(self, latent_dim, chem_dim, n_layers=6, hidden_dim=256):
        super(FlowRefiner, self).__init__()
        self.latent_dim = latent_dim
        self.chem_dim = chem_dim
        self.n_layers = n_layers
        self.coupling_layers = nn.ModuleList()
        for i in range(n_layers):
            self.coupling_layers.append(
                CouplingLayer(latent_dim, chem_dim,
                              hidden_dim, mask_type=i % 2)
            )
    def forward(self, z_ot, p):
        z = z_ot
        log_det_jac = 0
        for layer in self.coupling_layers:
            z, log_det = layer(z, p)
            log_det_jac += log_det
        return z, log_det_jac
class CouplingLayer(nn.Module):
    def __init__(self, latent_dim, chem_dim, hidden_dim, mask_type=0):
        super(CouplingLayer, self).__init__()
        self.latent_dim = latent_dim
        self.mask_type = mask_type
        mask = torch.zeros(latent_dim)
        mask[mask_type::2] = 1
        self.register_buffer('mask', mask)
        self.scale_net = nn.Sequential(
            nn.Linear(latent_dim // 2 + chem_dim, hidden_dim),
            nn.GELU(),
            nn.Linear(hidden_dim, latent_dim // 2),
            nn.Tanh()
        )
        self.translate_net = nn.Sequential(
            nn.Linear(latent_dim // 2 + chem_dim, hidden_dim),
            nn.GELU(),
            nn.Linear(hidden_dim, latent_dim // 2)
        )
    def forward(self, z, p):
        masked_z = z * self.mask
        unmasked_z = z * (1 - self.mask)
        z1, z2 = torch.chunk(unmasked_z, 2, dim=1)
        scale_input = torch.cat([z1, p], dim=1)
        scale = self.scale_net(scale_input)
        translate = self.translate_net(scale_input)
        z2_new = z2 * torch.exp(scale) + translate
        z_new = masked_z + torch.cat([z1, z2_new], dim=1) * (1 - self.mask)
        log_det_jac = scale.sum(dim=1)
        return z_new, log_det_jac
class GeneDecoder(nn.Module):
    def __init__(self, latent_dim, output_dim, hidden_dim=512, n_modules=10, dropout=0.1):
        super(GeneDecoder, self).__init__()
        self.latent_dim = latent_dim
        self.output_dim = output_dim
        self.n_modules = n_modules
        self.main_decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.module_heads = nn.ModuleList()
        module_size = output_dim // n_modules
        for i in range(n_modules):
            start_idx = i * module_size
            end_idx = start_idx + module_size if i < n_modules - 1 else output_dim
            head = nn.Sequential(
                nn.Linear(hidden_dim, hidden_dim // 2),
                nn.LayerNorm(hidden_dim // 2),
                nn.GELU(),
                nn.Dropout(dropout),
                nn.Linear(hidden_dim // 2, end_idx - start_idx)
            )
            self.module_heads.append(head)
        self.global_residual = nn.Linear(latent_dim, output_dim)
    def forward(self, z):
        main_feat = self.main_decoder(z)
```

```
        module_outputs = []
        for head in self.module_heads:
            module_outputs.append(head(main_feat))
        module_pred = torch.cat(module_outputs, dim=1)
        global_pred = self.global_residual(z)
        output = module_pred + 0.1 * global_pred
        return output
class CondOTGRNModel(nn.Module):
    def __init__(self, input_dim, output_dim, pert_dim, dose_dim=1, latent_dim=128, chem_dim=64,
                 hidden_dim=512, n_heads=8, n_layers=2, dropout=0.1,
                 ot_eps=0.1, ot_max_iter=50, flow_layers=6,
                 model_in_dim=128, bottleneck_dims=(2048, 512), use_bottleneck=True):
        super(CondOTGRNModel, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.model_in_dim = model_in_dim
        self.pert_dim = pert_dim
        self.dose_dim = dose_dim
        self.latent_dim = latent_dim
        self.chem_dim = chem_dim
        self.use_bottleneck = use_bottleneck
        if self.use_bottleneck:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, bottleneck_dims[0]),
                nn.LayerNorm(bottleneck_dims[0]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[0], bottleneck_dims[1]),
                nn.LayerNorm(bottleneck_dims[1]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[1], self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
            )
        else:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
                nn.GELU()
            )
        self.gene_encoder = GeneEncoder(
            self.model_in_dim, latent_dim, hidden_dim, n_heads, n_layers, dropout)
        self.chem_encoder = ChemEncoder(
            pert_dim, dose_dim, chem_dim, hidden_dim, dropout)
        self.conditional_ot = ConditionalOT(
            latent_dim, chem_dim, ot_eps, ot_max_iter)
        self.flow_refiner = FlowRefiner(
            latent_dim, chem_dim, flow_layers, hidden_dim)
        self.gene_decoder = GeneDecoder(
            latent_dim, self.model_in_dim, hidden_dim, n_modules=10, dropout=dropout)
        self.output_head = nn.Sequential(
            nn.Linear(self.model_in_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, self.output_dim)
        )
        self.apply(self._init_weights)
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)
    def forward(self, x_control, pert, dose, is_training=True):
        assert x_control.dim() == 2 and x_control.size(1) == self.input_dim, \
            f"Expected x_control shape [B, {self.input_dim}], got {x_control.shape}"
        x_control_proj = self.input_proj(x_control)
        z_control = self.gene_encoder(x_control_proj)
        p = self.chem_encoder(pert, dose)
        noise_scale = torch.norm(p, dim=1, keepdim=True) * 0.1
        noise = torch.randn_like(z_control) * noise_scale
        z_refined = z_control + noise
        z_refined, log_det_jac = self.flow_refiner(z_refined, p)
        x_pred_proj = self.gene_decoder(z_refined)
        # [OK] Changed from delta-based to absolute-based: directly predict absolute target value
        x_pred = self.output_head(x_pred_proj)
        x_pred_delta = x_pred - x_control
        return {
            'x_pred': x_pred,
            'x_pred_delta': x_pred_delta,
            'z_control': z_control,
            'z_refined': z_refined,
            'p': p,
            'log_det_jac': log_det_jac
        }
def train_model(model, train_loader, optimizer, scheduler, device, loss_weights=None):
    model.train()
    total_loss = 0
    accumulation_steps = 4
    optimizer.zero_grad()
    if loss_weights is None:
        loss_weights = {
            'recon': 1.0,
            'ot': 0.1,
            'flow': 0.1,
            'de': 0.5,
            'reg': 1e-5
        }
    for i, batch in enumerate(tqdm(train_loader, desc="Training")):
        x_baseline, pert, dose, x_target_abs = batch
        x_baseline, pert, dose, x_target_abs = x_baseline.to(
            device), pert.to(device), dose.to(device), x_target_abs.to(device)
```

```
        outputs = model(x_baseline, pert, dose, is_training=True)
        recon_loss = F.mse_loss(outputs['x_pred'], x_target_abs)
        with torch.no_grad():
            if x_target_abs.size(1) == model.input_dim:
                x_target_proj = model.input_proj(x_target_abs)
            else:
                x_target_proj = x_target_abs
            z_target = model.gene_encoder(x_target_proj)
        z_ot_pred, transport_plan = model.conditional_ot(
            outputs['z_control'], z_target.detach(), outputs['p'])
        cost_matrix = torch.cdist(outputs['z_control'], z_target, p=2) ** 2
        ot_loss = torch.sum(transport_plan * cost_matrix) / x_baseline.size(0)
        flow_loss = F.mse_loss(outputs['z_refined'], z_ot_pred.detach()) + 0.01 * outputs['log_det_jac'].pow(2).mean()
        epsilon = 1e-8
        lfc_true = torch.log2((x_target_abs + epsilon) / (x_baseline + epsilon))
        lfc_abs = torch.abs(lfc_true)
        K = min(20, x_target_abs.size(1))
        top_k_indices = torch.topk(lfc_abs, K, dim=1)[1]
        de_mask = torch.zeros_like(lfc_abs, dtype=torch.bool)
        for b in range(x_target_abs.size(0)):
            de_mask[b, top_k_indices[b]] = True
        x_target_delta = x_target_abs - x_baseline
        de_loss = F.mse_loss(outputs['x_pred_delta'][de_mask], x_target_delta[de_mask]) if de_mask.any() else torch.tensor(0.0, device=device)
        reg_loss = sum(p.pow(2.0).mean() for p in model.parameters())
        total_loss_batch = (loss_weights['recon'] * recon_loss +
                            loss_weights['ot'] * ot_loss +
                            loss_weights['flow'] * flow_loss +
                            loss_weights['de'] * de_loss +
                            loss_weights['reg'] * reg_loss)
        total_loss_batch = total_loss_batch / accumulation_steps
        total_loss_batch.backward()
        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()
        total_loss += total_loss_batch.item() * accumulation_steps
    return total_loss / len(train_loader)
def evaluate_model(model, test_loader, device, loss_weights=None):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    if loss_weights is None:
        loss_weights = {
            'recon': 1.0,
            'ot': 0.1,
            'flow': 0.1,
            'de': 0.5,
            'reg': 1e-5
        }
    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating"):
            x_baseline, pert, dose, x_target_abs = batch
            x_baseline, pert, dose, x_target_abs = x_baseline.to(
                device), pert.to(device), dose.to(device), x_target_abs.to(device)
            outputs = model(x_baseline, pert, dose, is_training=False)
            recon_loss = F.mse_loss(outputs['x_pred'], x_target_abs)
            with torch.no_grad():
                if x_target_abs.size(1) == model.input_dim:
                    x_target_proj = model.input_proj(x_target_abs)
                else:
                    x_target_proj = x_target_abs
                z_target = model.gene_encoder(x_target_proj)
            z_ot_pred, transport_plan = model.conditional_ot(
                outputs['z_control'], z_target.detach(), outputs['p'])
            cost_matrix = torch.cdist(outputs['z_control'], z_target, p=2) ** 2
            ot_loss = torch.sum(transport_plan * cost_matrix) / x_baseline.size(0)
            flow_loss = 0.01 * outputs['log_det_jac'].pow(2).mean()
            epsilon = 1e-8
            lfc_true = torch.log2((x_target_abs + epsilon) / (x_baseline + epsilon))
            lfc_abs = torch.abs(lfc_true)
            K = min(20, x_target_abs.size(1))
            top_k_indices = torch.topk(lfc_abs, K, dim=1)[1]
            de_mask = torch.zeros_like(lfc_abs, dtype=torch.bool)
            for b in range(x_target_abs.size(0)):
                de_mask[b, top_k_indices[b]] = True
            x_target_delta = x_target_abs - x_baseline
            de_loss = F.mse_loss(outputs['x_pred_delta'][de_mask], x_target_delta[de_mask]) if de_mask.any() else torch.tensor(0.0, device=device)
            reg_loss = sum(p.pow(2.0).mean() for p in model.parameters())
            loss = (loss_weights['recon'] * recon_loss +
                    loss_weights['ot'] * ot_loss +
                    loss_weights['flow'] * flow_loss +
                    loss_weights['de'] * de_loss +
                    loss_weights['reg'] * reg_loss)
            total_loss += loss.item()
            x_target_np = x_target_abs.cpu().numpy()
            x_pred_np = outputs['x_pred'].cpu().numpy()
            if not (np.any(np.isnan(x_target_np)) or np.any(np.isnan(x_pred_np)) or \
               np.any(np.isinf(x_target_np)) or np.any(np.isinf(x_pred_np))):
                all_targets.append(x_target_np)
                all_predictions.append(x_pred_np)
    if len(all_targets) > 0:
        all_targets = np.concatenate(all_targets, axis=0)
        all_predictions = np.concatenate(all_predictions, axis=0)
        try:
            r2 = r2_score(all_targets, all_predictions)
            if np.isnan(r2):
                r2 = 0.0
        except:
            r2 = 0.0
        try:
            pred_flat = all_predictions.flatten()
            true_flat = all_targets.flatten()
            if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
```

```
            pearson = pearsonr(true_flat, pred_flat)[0]
            if np.isnan(pearson):
                pearson = 0.0
        else:
            pearson = 0.0
    except:
        pearson = 0.0
else:
    r2 = 0.0
    pearson = 0.0
return {
    'loss': total_loss / len(test_loader),
    'r2': r2,
    'pearson': pearson
}
def calculate_detailed_metrics(pred, true, de_genes=None, control_baseline=None, perturbations=None):
    n_samples, n_genes = true.shape
    mse = np.mean((pred - true) ** 2)
    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0
    true_mean_vector = np.mean(true, axis=0)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_vector) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0
    if de_genes is not None:
        de_mask = np.zeros(n_genes, dtype=bool)
        de_mask[de_genes] = True
        if np.any(de_mask):
            mse_de = np.mean((pred[:, de_mask] - true[:, de_mask]) ** 2)
            true_de = true[:, de_mask]
            pred_de = pred[:, de_mask]
            true_de_mean_per_gene = np.mean(true_de, axis=0)
            pred_de_mean_per_gene = np.mean(pred_de, axis=0)
            true_de_centered = true_de - true_de_mean_per_gene
            pred_de_centered = pred_de - pred_de_mean_per_gene
            numerator_de = np.sum(true_de_centered * pred_de_centered)
            true_de_norm = np.sqrt(np.sum(true_de_centered ** 2))
            pred_de_norm = np.sqrt(np.sum(pred_de_centered ** 2))
            if true_de_norm > 1e-10 and pred_de_norm > 1e-10:
                pcc_de = numerator_de / (true_de_norm * pred_de_norm)
                if np.isnan(pcc_de):
                    pcc_de = 0.0
            else:
                pcc_de = 0.0
            true_de_mean_vector = np.mean(true_de, axis=0)
            ss_res_de = np.sum((true_de - pred_de) ** 2)
            ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
            if ss_tot_de > 1e-10:
                r2_de = 1.0 - (ss_res_de / ss_tot_de)
            else:
                r2_de = 0.0
            if np.isnan(r2_de):
                r2_de = 0.0
        else:
            mse_de = pcc_de = r2_de = np.nan
    elif control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) / (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
                    pred_de_flat = pred_de.flatten()
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_vector = np.mean(true_de, axis=0)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
```

```
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                mse_de = np.mean((pred_de - true_de) ** 2)
                pred_de_flat = pred_de.flatten()
                true_de_flat = true_de.flatten()
                if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                    pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0
                true_de_mean_vector = np.mean(true_de, axis=0)
                ss_res_de = np.sum((true_de - pred_de) ** 2)
                ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                if ss_tot_de > 1e-10:
                    r2_de = 1.0 - (ss_res_de / ss_tot_de)
                else:
                    r2_de = 0.0
                if np.isnan(r2_de):
                    r2_de = 0.0
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }
def objective(trial, timestamp):
    global train_dataset, test_dataset, device, scaler, test_adata
    params = {
        'latent_dim': trial.suggest_categorical('latent_dim', [64, 128, 256]),
        'chem_dim': trial.suggest_categorical('chem_dim', [32, 64, 128]),
        'hidden_dim': trial.suggest_categorical('hidden_dim', [256, 512, 768, 1024]),
        'n_layers': trial.suggest_int('n_layers', 2, 4),
        'n_heads': trial.suggest_categorical('n_heads', [4, 8, 16]),
        'dropout': trial.suggest_float('dropout', 0.1, 0.3),
        'ot_eps': trial.suggest_float('ot_eps', 0.05, 0.2),
        'ot_max_iter': trial.suggest_int('ot_max_iter', 30, 100),
        'flow_layers': trial.suggest_int('flow_layers', 4, 8),
        'learning_rate': trial.suggest_float('learning_rate', 1e-4, 1e-3, log=True),
        'weight_decay': trial.suggest_float('weight_decay', 1e-5, 1e-4, log=True),
        'batch_size': trial.suggest_categorical('batch_size', [32, 64, 128, 256]),
        'alpha': trial.suggest_float('alpha', 0.5, 2.0),
        'beta': trial.suggest_float('beta', 0.1, 1.0),
        'gamma': trial.suggest_float('gamma', 0.05, 0.2),
        'eta': trial.suggest_float('eta', 0.1, 1.0)
    }
    while params['hidden_dim'] % params['n_heads'] != 0:
        if params['hidden_dim'] < 1024:
            params['hidden_dim'] += params['n_heads']
        else:
            params['hidden_dim'] = 512
    test_perturbation_names = list(test_adata.obs['perturbation'].unique())
    max_pert_dim, all_pert_names = standardize_perturbation_encoding(
        train_dataset, test_dataset=None, test_perturbation_names=test_perturbation_names)
    n_genes = train_dataset.n_genes
    print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
    model = CondOTGRNModel(
        input_dim=n_genes,
        output_dim=n_genes,
        pert_dim=max_pert_dim,
        dose_dim=1,
        latent_dim=params['latent_dim'],
        chem_dim=params['chem_dim'],
        hidden_dim=params['hidden_dim'],
        n_layers=params['n_layers'],
        n_heads=params['n_heads'],
        dropout=params['dropout'],
        ot_eps=params['ot_eps'],
        ot_max_iter=params['ot_max_iter'],
        flow_layers=params['flow_layers']
    ).to(device)
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=params['learning_rate'],
        weight_decay=params['weight_decay']
    )
    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer, T_0=10, T_mult=2, eta_min=1e-6
```

```
    )
    train_size = int(0.8 * len(train_dataset))
    val_size = len(train_dataset) - train_size
    train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
    train_loader = DataLoader(
        train_subset,
        batch_size=params['batch_size'],
        shuffle=True,
        num_workers=4,
        pin_memory=True
    )
    val_loader = DataLoader(
        val_subset,
        batch_size=params['batch_size'],
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
    best_val_loss = float('inf')
    patience = 15
    patience_counter = 0
    max_epochs = 100
    loss_weights = {
        'recon': params['alpha'],
        'ot': params['beta'],
        'flow': params['gamma'],
        'de': params['eta'],
        'grn': 0.01,
        'reg': 1e-5
    }
    for epoch in range(max_epochs):
        train_loss = train_model(
            model, train_loader, optimizer, scheduler, device, loss_weights)
        val_metrics = evaluate_model(model, val_loader, device, loss_weights)
        if val_metrics['loss'] < best_val_loss:
            best_val_loss = val_metrics['loss']
            patience_counter = 0
            torch.save(model.state_dict(),
                       f'best_model_trial_{trial.number}_{timestamp}.pt')
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break
        trial.report(val_metrics['loss'], epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()
    return best_val_loss
def evaluate_and_save_model(model, test_loader, device, save_path, common_genes_info=None, scaler=None, best_params=None):
    model.eval()
    all_predictions = []
    all_targets = []
    all_perturbations = []
    all_baselines = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc='Evaluating'):
            x_baseline, pert, dose, x_target_abs = batch
            x_baseline, pert, dose, x_target_abs = x_baseline.to(
                device), pert.to(device), dose.to(device), x_target_abs.to(device)
            outputs = model(x_baseline, pert, dose, is_training=False)
            all_predictions.append(outputs['x_pred'].cpu().numpy())
            all_targets.append(x_target_abs.cpu().numpy())
            all_perturbations.append(pert.cpu().numpy())
            all_baselines.append(x_baseline.cpu().numpy())
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)
    all_perturbations = np.concatenate(all_perturbations, axis=0)
    all_baselines = np.concatenate(all_baselines, axis=0)
    control_baseline = all_baselines
    results = calculate_detailed_metrics(all_predictions, all_targets, control_baseline=control_baseline, perturbations=all_perturbations)
    torch.save({
        'model_state_dict': model.state_dict(),
        'evaluation_results': results,
        'predictions': all_predictions,
        'targets': all_targets,
        'perturbations': all_perturbations,
        'baselines': all_baselines,
        'gene_names': common_genes_info['genes'] if common_genes_info is not None else None,
        'common_genes_info': common_genes_info,
        'scaler': scaler,
        'best_params': best_params,
        'model_config': {
            'input_dim': model.input_dim,
            'pert_dim': train_dataset.perturbations.shape[1],
            'dose_dim': 1,
            'latent_dim': model.latent_dim,
            'chem_dim': model.chem_dim,
            'hidden_dim': model.gene_encoder.input_proj.out_features,
            'n_layers': len([m for m in model.gene_encoder.mlp if isinstance(m, nn.Linear)]) // 2 if hasattr(model.gene_encoder, 'mlp') else 2,
            'n_heads': 8,
            'dropout': model.gene_encoder.output_proj[3].p if len(model.gene_encoder.output_proj) > 3 else 0.1,
            'ot_eps': model.conditional_ot.eps,
            'ot_max_iter': model.conditional_ot.max_iter,
            'flow_layers': len(model.flow_refiner.coupling_layers)
        }
    }, save_path)
    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']]
    })
    print("\nEvaluation Results:")
    print(metrics_df.to_string(index=False,
          float_format=lambda x: '{:.6f}'.format(x)))
    print(f"\nModel and evaluation results saved to: {save_path}")
    return results
```

```
def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(list(set(train_dataset.perturbation_names + test_dataset.perturbation_names)))
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []
    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))
    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(
        columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)
    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(np.float32)
        test_dataset._pert_encoding_standardized = True
    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True
    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names
def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print_log(f'Training started at: {timestamp}')
    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print_log(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print_log(
                    f'Warning: Specified GPU {gpu_id} does not exist, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print_log(f'Using specified GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print_log(f'Using default GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print_log('CUDA not available, using CPU')
    print_log('Loading data...')
    train_path = "/datasets/SrivatsanTrapnell2020_train_filtered2.h5ad"
    test_path = "/datasets/SrivatsanTrapnell2020_test_filtered2.h5ad"
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(
            f"Data files not found: {train_path} or {test_path}")
    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print_log(f'Training data shape: {train_adata.shape}')
    print_log(f'Test data shape: {test_adata.shape}')
    print_log("Processing gene set consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print_log(f"Training genes: {len(train_genes)}")
    print_log(f"Test genes: {len(test_genes)}")
    print_log(f"Common genes: {len(common_genes)}")
    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(
        gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(
        gene) for gene in common_genes]
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    scaler = StandardScaler()
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0
    print_log("=" * 60)
    print_log("CRITICAL: Training in FULL GENE SPACE")
    print_log(f"Gene count: {len(common_genes)}")
    print_log("Model will use input_proj to project to internal working dimension")
    print_log("=" * 60)
    common_genes_info = {
        'genes': common_genes,
        'train_idx': train_gene_idx,
```

```
        'test_idx': test_gene_idx
    }
    train_pert_col = train_adata.obs['perturbation'].astype(str)
    train_pert_counts = train_pert_col.value_counts()
    has_control = train_pert_col.str.lower().isin(['control', 'ctrl']).any() or \
                  train_pert_col.str.contains('NegCtrl', case=False, na=False).any()
    reference_pert = None
    if not has_control:
        reference_pert = train_pert_counts.index[0]
    train_dataset = GeneExpressionDataset(
        train_adata,
        perturbation_key='perturbation',
        dose_key='dose_value',
        scaler=scaler,
        augment=True,
        is_train=True,
        common_genes_info=common_genes_info,
        reference_perturbation=reference_pert
    )
    test_dataset = GeneExpressionDataset(
        test_adata,
        perturbation_key='perturbation',
        dose_key='dose_value',
        scaler=scaler,
        augment=False,
        is_train=False,
        common_genes_info=common_genes_info,
        reference_perturbation=reference_pert
    )
    optuna.logging.set_verbosity(optuna.logging.WARNING)
    study = optuna.create_study(
        direction='minimize',
        sampler=optuna.samplers.TPESampler(seed=42),
        pruner=optuna.pruners.MedianPruner(
            n_startup_trials=5,
            n_warmup_steps=10,
            interval_steps=1
        )
    )
    print_log('Starting hyperparameter optimization...')
    trials = 50
    with tqdm(total=trials, desc="Hyperparameter Optimization") as pbar:
        def objective_with_progress(trial):
            result = objective(trial, timestamp)
            pbar.update(1)
            pbar.set_postfix({
                'trial': trial.number,
                'value': f"{result:.4f}" if result is not None else "pruned"
            })
            return result
        study.optimize(objective_with_progress, n_trials=trials)
    max_pert_dim, all_pert_names = standardize_perturbation_encoding(
        train_dataset, test_dataset)
    n_genes = train_dataset.n_genes
    print_log(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
    best_params = study.best_params
    final_model = CondOTGRNModel(
        input_dim=n_genes,
        output_dim=n_genes,
        pert_dim=max_pert_dim,
        dose_dim=1,
        latent_dim=best_params['latent_dim'],
        chem_dim=best_params['chem_dim'],
        hidden_dim=best_params['hidden_dim'],
        n_layers=best_params['n_layers'],
        n_heads=best_params['n_heads'],
        dropout=best_params['dropout'],
        ot_eps=best_params['ot_eps'],
        ot_max_iter=best_params['ot_max_iter'],
        flow_layers=best_params['flow_layers']
    ).to(device)
    train_size = int(0.8 * len(train_dataset))
    val_size = len(train_dataset) - train_size
    train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
    train_loader = DataLoader(
        train_subset,
        batch_size=best_params['batch_size'],
        shuffle=True,
        num_workers=4,
        pin_memory=True
    )
    val_loader = DataLoader(
        val_subset,
        batch_size=best_params['batch_size'],
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
    test_loader = DataLoader(
        test_dataset,
        batch_size=best_params['batch_size'],
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
    optimizer = torch.optim.AdamW(
        final_model.parameters(),
        lr=best_params['learning_rate'],
        weight_decay=best_params['weight_decay']
    )
    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer, T_0=10, T_mult=2, eta_min=1e-6
    )
    print_log('Training final model...')
    best_loss = float('inf')
```

```
    best_model = None
    max_epochs = 200
    patience = 20
    patience_counter = 0
    loss_weights = {
        'recon': best_params['alpha'],
        'ot': best_params['beta'],
        'flow': best_params['gamma'],
        'de': best_params['eta'],
        'grn': 0.01,
        'reg': 1e-5
    }
    for epoch in range(max_epochs):
        train_loss = train_model(
            final_model, train_loader, optimizer, scheduler, device, loss_weights)
        val_metrics = evaluate_model(
            final_model, val_loader, device, loss_weights)
        if (epoch + 1) % 20 == 0:
            print_log(f'Epoch {epoch+1}/{max_epochs}:')
            print_log(f'Training Loss: {train_loss:.4f}')
            print_log(f'Validation Loss: {val_metrics["loss"]:.4f}')
        if val_metrics["loss"] < best_loss:
            best_loss = val_metrics["loss"]
            patience_counter = 0
            best_model = final_model.state_dict()
            torch.save({
                'epoch': epoch,
                'model_state_dict': final_model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss': best_loss,
                'metrics': val_metrics,
                'best_params': best_params,
                'scaler': scaler,
                'common_genes_info': common_genes_info
            }, f'condot_grn_best_model_{timestamp}.pt')
            print_log(f"Saved best model with validation loss: {best_loss:.4f}")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print_log(f'Early stopping at epoch {epoch+1}')
                break
    final_model.load_state_dict(best_model)
    print_log('Evaluating final model on test set...')
    results = evaluate_and_save_model(
        final_model, test_loader, device,
        f'condot_grn_final_model_{timestamp}.pt',
        common_genes_info, scaler, best_params
    )
    best_params_str = str(best_params)
    if len(best_params_str) > 50:
        best_params_str = best_params_str[:50] + '...'
    results_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']],
        'Best_Params': [best_params_str] * 6
    })
    results_df.to_csv(
        f'condot_grn_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(index=False,
          float_format=lambda x: '{:.6f}'.format(x)))
    return results_df
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='CondOT-GRN Model Training for Drug Perturbation Prediction')
    parser.add_argument('--gpu', type=int, default=None,
                        help='Specify GPU ID to use (0-7, e.g., --gpu 0)')
    parser.add_argument('--list-gpus', action='store_true',
                        help='List all available GPUs and exit')
    parser.add_argument('--epochs', type=int, default=200,
                        help='Number of training epochs (default: 200)')
    parser.add_argument('--trials', type=int, default=50,
                        help='Number of hyperparameter optimization trials (default: 50)')
    args = parser.parse_args()
    if args.list_gpus:
        if torch.cuda.is_available():
            gpu_count = torch.cuda.device_count()
            for i in range(gpu_count):
                print(f'GPU {i}: {torch.cuda.get_device_name(i)}')
        else:
            print('CUDA not available')
        exit(0)
    print_log("=" * 80)
    print_log("CondOT-GRN Model for Single-Cell Drug Perturbation Prediction")
    print_log("=" * 80)
    print_log(f"Epochs: {args.epochs}")
    print_log(f"Hyperparameter optimization trials: {args.trials}")
    if args.gpu is not None:
        print_log(f"Using specified GPU: {args.gpu}")
    results_df = main(gpu_id=args.gpu)
    print_log("Training completed successfully!")
    print_log("=" * 80)
```

D.4 *CPA-Traj* for Schiebinger cytokine time course (scRNA-seq)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
```

```
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm
import os
import pandas as pd
from scipy.stats import pearsonr
import time
from datetime import datetime
import argparse
import random
import copy


def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")


def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


train_adata = None
train_dataset = None
test_dataset = None
device = None
pca_model = None
scaler = None


class CytokineTrajectoryDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', age_key='age',
                 scaler=None, pca_model=None, pca_dim=128, fit_pca=False,
                 augment=False, is_train=True, common_genes_info=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.age_key = age_key
        self.augment = augment
        self.is_train = is_train
        self.pca_dim = pca_dim
        self.common_genes_info = common_genes_info

        # ===== Gene subset & preprocessing =====
        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X

        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)

        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)

        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (
            abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)

        if not is_log1p_already:
            data = np.log1p(data)

        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            if not is_standardized_already:
                data = self.scaler.transform(data)

        data = np.clip(data, -10, 10)
        data = data / 10.0

        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self.pca = pca_model if pca_model is not None else None
        self.pca_dim = pca_dim if pca_model is not None else None

        # ===== Perturbation encoding =====
        self.perturbation_names = [
```

```
            x for x in adata.obs[perturbation_key].unique() if pd.notnull(x)]
        self.perturbations = pd.get_dummies(
            adata.obs[perturbation_key]).values.astype(np.float32)
        print_log(
            f"{'Training' if is_train else 'Test/Val'} set perturbation dimension: {self.perturbations.shape[1]}")

        perturbation_labels = adata.obs[perturbation_key].astype(str).values
        perturbation_labels = np.array(perturbation_labels, dtype='U')
        lower_labels = np.char.lower(perturbation_labels)
        negctrl_mask = np.char.find(lower_labels, 'negctrl') >= 0
        control_mask = (lower_labels == 'control') | (
            lower_labels == 'ctrl') | negctrl_mask
        self._control_indices = np.where(control_mask)[0]
        self._non_control_indices = np.where(~control_mask)[0]
        self._control_set = set(self._control_indices.tolist())

        self._non_control_pert_names = [
            name for name in self.perturbation_names
            if name not in ['control', 'Control', 'ctrl', 'Ctrl']
            and 'negctrl' not in name.lower()
        ]

        if len(self._control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
                    "This model performs control -> perturbed prediction, NOT autoencoder. "
                    "Please ensure your training data includes control samples "
                    "(labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                raise ValueError(
                    "CRITICAL ERROR: Evaluation set has NO control samples! "
                    "For unseen perturbation prediction, evaluation set should use training set control baseline. "
                    "Please ensure training set has control samples."
                )

        self.rng = np.random.RandomState(42)

        # ===== Time embeddings =====
        self.time_embeddings = self._encode_timepoints(adata.obs[age_key])
        print_log(
            f"{'Training' if is_train else 'Test/Val'} set time dimension: {self.time_embeddings.shape[1]}")

        # ? is_train=False ? dataset?val/test????? baseline-target pair
        if not self.is_train:
            self._create_fixed_pairs_for_test()

    def _encode_timepoints(self, timepoints):
        """
        Encode timepoints like 'D0', 'D2', 'iPSC' into a 3D embedding: [sin, cos, normalized].
        """
        time_mapping = {}
        unique_times = sorted(timepoints.unique())
        for raw_tp in unique_times:
            time_str = str(raw_tp)
            if time_str in ('iPSC', 'iPSCs'):
                time_mapping[time_str] = 20.0
            elif time_str.startswith('D'):
                try:
                    time_mapping[time_str] = float(time_str[1:])
                except Exception:
                    time_mapping[time_str] = 0.0
            else:
                time_mapping[time_str] = 0.0

        time_features = []
        for tp in timepoints:
            time_val = time_mapping[str(tp)]
            sin_time = np.sin(2 * np.pi * time_val / 20.0)
            cos_time = np.cos(2 * np.pi * time_val / 20.0)
            norm_time = time_val / 20.0
            time_features.append([sin_time, cos_time, norm_time])
        return np.array(time_features, dtype=np.float32)

    def _create_fixed_pairs_for_test(self):
        """
        For evaluation datasets, create deterministic (baseline_idx, target_idx) pairs.
        """
        n = len(self.adata)
        self.pairs = [None] * n
        for idx in range(n):
            if idx in self._control_set:
                baseline_idx = idx
                if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                    target_idx = int(
                        self._non_control_indices[idx % len(self._non_control_indices)])
                else:
                    alternative_controls = self._control_indices[self._control_indices != idx]
                    if len(alternative_controls) > 0:
                        target_idx = int(
                            alternative_controls[idx % len(alternative_controls)])
                    else:
                        target_idx = idx
            else:
                if len(self._control_indices) > 0:
                    baseline_idx = int(
                        self._control_indices[idx % len(self._control_indices)])
                else:
                    baseline_idx = idx
                target_idx = idx
            self.pairs[idx] = (baseline_idx, target_idx)

    def __len__(self):
```

```
        return len(self.adata)

    def __getitem__(self, idx):
        x_current = self.expression_data[idx]
        pert = self.perturbations[idx]
        time_emb = self.time_embeddings[idx]

        if self.is_train:
            # ???????? baseline-target
            if idx in self._control_set:
                x_baseline = x_current
                if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                    target_idx = int(self.rng.choice(
                        self._non_control_indices))
                    x_target = self.original_expression_data[target_idx]
                    pert_target = self.perturbations[target_idx]
                else:
                    alternative_controls = self._control_indices[self._control_indices != idx]
                    if len(alternative_controls) > 0:
                        target_idx = int(self.rng.choice(alternative_controls))
                        x_target = self.original_expression_data[target_idx]
                        pert_target = self.perturbations[target_idx]
                    else:
                        x_target = self.original_expression_data[idx]
                        pert_target = pert
            else:
                if len(self._control_indices) > 0:
                    baseline_idx = int(self.rng.choice(self._control_indices))
                    x_baseline = self.expression_data[baseline_idx]
                else:
                    x_baseline = x_current
                x_target = self.original_expression_data[idx]
                pert_target = pert
        else:
            # ??/???????? baseline-target pair
            baseline_idx, target_idx = self.pairs[idx]
            if baseline_idx == idx and idx in self._control_set:
                x_baseline = x_current
            else:
                x_baseline = self.expression_data[baseline_idx]
            x_target = self.original_expression_data[target_idx]
            if target_idx == idx:
                pert_target = pert
            else:
                pert_target = self.perturbations[target_idx]

        x_target_delta = x_target - x_baseline

        # ??????????
        if self.augment and self.is_train:
            noise = np.random.normal(0, 0.05, x_baseline.shape)
            x_baseline = x_baseline + noise
            mask = np.random.random(x_baseline.shape) > 0.05
            x_baseline = x_baseline * mask

        return (
            torch.FloatTensor(x_baseline),
            torch.FloatTensor(pert_target),
            torch.FloatTensor(time_emb),
            torch.FloatTensor(x_target_delta)
        )


class TrajectoryAwareEncoder(nn.Module):
    def __init__(self, input_dim, latent_dim, hidden_dim, time_dim=3):
        super().__init__()
        self.input_dim = input_dim
        self.latent_dim = latent_dim
        self.time_dim = time_dim

        self.input_proj = nn.Linear(input_dim, hidden_dim)
        self.time_embedding = nn.Linear(time_dim, hidden_dim)

        mlp_layers = []
        for _ in range(2):
            mlp_layers.extend([
                nn.Linear(hidden_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(0.1)
            ])
        self.shared_backbone = nn.Sequential(
            *mlp_layers) if mlp_layers else nn.Identity()

        self.shared_mu = nn.Linear(hidden_dim, latent_dim // 2)
        self.shared_logvar = nn.Linear(hidden_dim, latent_dim // 2)
        self.condition_mu = nn.Linear(hidden_dim, latent_dim // 2)
        self.condition_logvar = nn.Linear(hidden_dim, latent_dim // 2)

    def forward(self, x, time_emb):
        x_proj = self.input_proj(x)               # [B, H]
        time_proj = self.time_embedding(time_emb)  # [B, H]
        fused = x_proj + time_proj
        encoded = self.shared_backbone(fused)

        shared_mu = self.shared_mu(encoded)
        shared_logvar = self.shared_logvar(encoded)
        condition_mu = self.condition_mu(encoded)
        condition_logvar = self.condition_logvar(encoded)

        shared_std = torch.exp(0.5 * shared_logvar)
        shared_eps = torch.randn_like(shared_std)
        shared_z = shared_mu + shared_eps * shared_std

        condition_std = torch.exp(0.5 * condition_logvar)
```

```
        condition_eps = torch.randn_like(condition_std)
        condition_z = condition_mu + condition_eps * condition_std

        z = torch.cat([shared_z, condition_z], dim=1)  # [B, latent_dim]
        return z, shared_mu, shared_logvar, condition_mu, condition_logvar


class PerturbationMixingModule(nn.Module):
    def __init__(self, latent_dim, pert_dim, hidden_dim=256):
        super().__init__()
        self.latent_dim = latent_dim
        self.pert_dim = pert_dim

        self.pert_embedding = nn.Linear(pert_dim, hidden_dim)
        self.mixing_layers = nn.ModuleList([
            nn.Sequential(
                nn.Linear(hidden_dim + latent_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(0.1)
            ) for _ in range(3)
        ])
        self.output_proj = nn.Linear(hidden_dim, latent_dim)

    def forward(self, z, pert):
        pert_emb = self.pert_embedding(pert)   # [B, H]
        x = torch.cat([z, pert_emb], dim=1)    # [B, latent+H]

        for layer in self.mixing_layers:
            x = layer(x)
            x = torch.cat([z, x], dim=1)       # maintain [B, latent+H]

        mixed_features = x[:, self.latent_dim:]  # take the last H dims
        output = self.output_proj(mixed_features)  # [B, latent]
        return output


class GraphRegularizedDecoder(nn.Module):
    def __init__(self, latent_dim, output_dim, hidden_dim=512):
        super().__init__()
        self.latent_dim = latent_dim
        self.output_dim = output_dim

        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(hidden_dim, output_dim)
        )

        # Graph regularizer ?????????????? loss ???? delta_expr ????
        self.graph_regularizer = nn.Linear(output_dim, output_dim, bias=False)

    def forward(self, z):
        delta_expr = self.decoder(z)                  # [B, G]
        regularized = self.graph_regularizer(delta_expr)
        return delta_expr, regularized


class CytokineTrajectoryModel(nn.Module):
    def __init__(self, input_dim, output_dim, pert_dim, time_dim=3, latent_dim=128,
                 hidden_dim=512, model_in_dim=128, bottleneck_dims=(2048, 512),
                 use_bottleneck=True):
        super(CytokineTrajectoryModel, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.model_in_dim = model_in_dim
        self.pert_dim = pert_dim
        self.time_dim = time_dim
        self.latent_dim = latent_dim
        self.hidden_dim = hidden_dim
        self.use_bottleneck = use_bottleneck
        self.bottleneck_dims = bottleneck_dims

        if self.use_bottleneck:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, bottleneck_dims[0]),
                nn.LayerNorm(bottleneck_dims[0]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[0], bottleneck_dims[1]),
                nn.LayerNorm(bottleneck_dims[1]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[1], self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
            )
        else:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
                nn.GELU()
            )

        self.encoder = TrajectoryAwareEncoder(
            input_dim=self.model_in_dim,
            latent_dim=latent_dim,
            hidden_dim=hidden_dim,
            time_dim=time_dim
        )
```

```
        self.perturbation_mixing = PerturbationMixingModule(
            latent_dim=latent_dim,
            pert_dim=pert_dim,
            hidden_dim=hidden_dim
        )

        self.decoder = GraphRegularizedDecoder(
            latent_dim=latent_dim,
            output_dim=output_dim,
            hidden_dim=hidden_dim
        )

        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)

    def forward(self, x, pert, time_emb):
        assert x.dim() == 2 and x.size(1) == self.input_dim, \
            f"Expected x shape [B, {self.input_dim}], got {x.shape}"

        x_proj = self.input_proj(x)  # [B, model_in_dim]
        z, shared_mu, shared_logvar, condition_mu, condition_logvar = self.encoder(
            x_proj, time_emb)

        z_mixed = self.perturbation_mixing(z, pert)
        predicted_expr, regularized_expr = self.decoder(z_mixed)
        delta_expr = predicted_expr  # decoder output is already delta

        return {
            'predicted_expr': predicted_expr,
            'delta_expr': delta_expr,
            'regularized_expr': regularized_expr,
            'shared_mu': shared_mu,
            'shared_logvar': shared_logvar,
            'condition_mu': condition_mu,
            'condition_logvar': condition_logvar,
            'latent_z': z
        }


def train_model(model, train_loader, optimizer, scheduler, device,
                kl_weight=0.1, graph_weight=0.01):
    model.train()
    total_loss = 0.0
    accumulation_steps = 4
    optimizer.zero_grad()

    for i, batch in enumerate(tqdm(train_loader, desc="Training", leave=False)):
        x_baseline, pert, time_emb, x_target_delta = batch
        x_baseline = x_baseline.to(device)
        pert = pert.to(device)
        time_emb = time_emb.to(device)
        x_target_delta = x_target_delta.to(device)

        outputs = model(x_baseline, pert, time_emb)

        recon_loss = F.mse_loss(outputs['predicted_expr'], x_target_delta)

        shared_kl = -0.5 * torch.sum(
            1 + outputs['shared_logvar'] -
            outputs['shared_mu'].pow(2) -
            outputs['shared_logvar'].exp(), dim=1
        ).mean()

        condition_kl = -0.5 * torch.sum(
            1 + outputs['condition_logvar'] -
            outputs['condition_mu'].pow(2) -
            outputs['condition_logvar'].exp(), dim=1
        ).mean()

        kl_loss = shared_kl + condition_kl

        delta_expr = outputs['delta_expr']
        delta_mean = delta_expr.mean(dim=1, keepdim=True)
        laplacian_loss = torch.mean(
            (delta_expr - delta_mean).pow(2).sum(dim=1))
        graph_loss = laplacian_loss

        loss = recon_loss + kl_weight * kl_loss + graph_weight * graph_loss
        loss = loss / accumulation_steps
        loss.backward()

        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

        total_loss += loss.item() * accumulation_steps

    return total_loss / len(train_loader)


def evaluate_model(model, test_loader, device,
                   kl_weight=0.1, graph_weight=0.01):
    model.eval()
    total_loss = 0.0
```

```
    all_targets = []
    all_predictions = []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc="Evaluating", leave=False):
            x_baseline, pert, time_emb, x_target_delta = batch
            x_baseline = x_baseline.to(device)
            pert = pert.to(device)
            time_emb = time_emb.to(device)
            x_target_delta = x_target_delta.to(device)

            outputs = model(x_baseline, pert, time_emb)

            recon_loss = F.mse_loss(outputs['predicted_expr'], x_target_delta)

            shared_kl = -0.5 * torch.sum(
                1 + outputs['shared_logvar'] -
                outputs['shared_mu'].pow(2) -
                outputs['shared_logvar'].exp(), dim=1
            ).mean()

            condition_kl = -0.5 * torch.sum(
                1 + outputs['condition_logvar'] -
                outputs['condition_mu'].pow(2) -
                outputs['condition_logvar'].exp(), dim=1
            ).mean()

            kl_loss = shared_kl + condition_kl

            delta_expr = outputs['delta_expr']
            delta_mean = delta_expr.mean(dim=1, keepdim=True)
            laplacian_loss = torch.mean(
                (delta_expr - delta_mean).pow(2).sum(dim=1))
            graph_loss = laplacian_loss

            loss = recon_loss + kl_weight * kl_loss + graph_weight * graph_loss
            total_loss += loss.item()

            x_target_abs = x_baseline + x_target_delta
            x_pred_abs = x_baseline + outputs['predicted_expr']
            all_targets.append(x_target_abs.cpu().numpy())
            all_predictions.append(x_pred_abs.cpu().numpy())

    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)

    true_mean_global = np.mean(all_targets)
    ss_res = np.sum((all_targets - all_predictions) ** 2)
    ss_tot = np.sum((all_targets - true_mean_global) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0

    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pear = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pear):
            pear = 0.0
    else:
        pear = 0.0

    return {
        'loss': total_loss / len(test_loader),
        'r2': r2,
        'pearson': pear
    }


def calculate_detailed_metrics(pred, true, de_genes=None, control_baseline=None, perturbations=None):
    n_samples, n_genes = true.shape

    mse = np.mean((pred - true) ** 2)

    # global PCC
    true_mean_per_gene = np.mean(true, axis=0)
    pred_mean_per_gene = np.mean(pred, axis=0)
    true_centered = true - true_mean_per_gene
    pred_centered = pred - pred_mean_per_gene
    numerator = np.sum(true_centered * pred_centered)
    true_norm = np.sqrt(np.sum(true_centered ** 2))
    pred_norm = np.sqrt(np.sum(pred_centered ** 2))
    if true_norm > 1e-10 and pred_norm > 1e-10:
        pcc = numerator / (true_norm * pred_norm)
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0

    # global R2
    true_mean_vector = np.mean(true, axis=0)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_vector) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0

    # DE metrics
    if de_genes is not None:
```

```
        de_mask = np.zeros(n_genes, dtype=bool)
        de_mask[de_genes] = True
        if np.any(de_mask):
            mse_de = np.mean((pred[:, de_mask] - true[:, de_mask]) ** 2)
            true_de = true[:, de_mask]
            pred_de = pred[:, de_mask]

            true_de_mean_per_gene = np.mean(true_de, axis=0)
            pred_de_mean_per_gene = np.mean(pred_de, axis=0)
            true_de_centered = true_de - true_de_mean_per_gene
            pred_de_centered = pred_de - pred_de_mean_per_gene
            numerator_de = np.sum(true_de_centered * pred_de_centered)
            true_de_norm = np.sqrt(np.sum(true_de_centered ** 2))
            pred_de_norm = np.sqrt(np.sum(pred_de_centered ** 2))
            if true_de_norm > 1e-10 and pred_de_norm > 1e-10:
                pcc_de = numerator_de / (true_de_norm * pred_de_norm)
                if np.isnan(pcc_de):
                    pcc_de = 0.0
            else:
                pcc_de = 0.0

            true_de_mean_vector = np.mean(true_de, axis=0)
            ss_res_de = np.sum((true_de - pred_de) ** 2)
            ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
            if ss_tot_de > 1e-10:
                r2_de = 1.0 - (ss_res_de / ss_tot_de)
            else:
                r2_de = 0.0
            if np.isnan(r2_de):
                r2_de = 0.0
        else:
            mse_de = pcc_de = r2_de = np.nan
    elif control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []

            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) /
                              (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True

                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)

                    true_de_mean_per_gene = np.mean(true_de, axis=0)
                    pred_de_mean_per_gene = np.mean(pred_de, axis=0)
                    true_de_centered = true_de - true_de_mean_per_gene
                    pred_de_centered = pred_de - pred_de_mean_per_gene
                    numerator_de = np.sum(true_de_centered * pred_de_centered)
                    true_de_norm = np.sqrt(np.sum(true_de_centered ** 2))
                    pred_de_norm = np.sqrt(np.sum(pred_de_centered ** 2))
                    if true_de_norm > 1e-10 and pred_de_norm > 1e-10:
                        pcc_de_pert = numerator_de / \
                            (true_de_norm * pred_de_norm)
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0

                    true_de_mean_vector = np.mean(true_de, axis=0)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0

                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)

            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
```

```
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                mse_de = np.mean((pred_de - true_de) ** 2)

                true_de_mean_per_gene = np.mean(true_de, axis=0)
                pred_de_mean_per_gene = np.mean(pred_de, axis=0)
                true_de_centered = true_de - true_de_mean_per_gene
                pred_de_centered = pred_de - pred_de_mean_per_gene
                numerator_de = np.sum(true_de_centered * pred_de_centered)
                true_de_norm = np.sqrt(np.sum(true_de_centered ** 2))
                pred_de_norm = np.sqrt(np.sum(pred_de_centered ** 2))
                if true_de_norm > 1e-10 and pred_de_norm > 1e-10:
                    pcc_de = numerator_de / (true_de_norm * pred_de_norm)
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0

                true_de_mean_vector = np.mean(true_de, axis=0)
                ss_res_de = np.sum((true_de - pred_de) ** 2)
                ss_tot_de = np.sum((true_de - true_de_mean_vector) ** 2)
                if ss_tot_de > 1e-10:
                    r2_de = 1.0 - (ss_res_de / ss_tot_de)
                else:
                    r2_de = 0.0
                if np.isnan(r2_de):
                    r2_de = 0.0
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan

    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }


def evaluate_and_save_model(model, test_loader, device, save_path,
                            common_genes_info=None, pca_model=None, scaler=None, best_params=None):
    model.eval()
    all_predictions = []
    all_targets = []
    all_baselines = []
    all_perts = []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc='Evaluating'):
            x_baseline, pert, time_emb, x_target_delta = batch
            x_baseline = x_baseline.to(device)
            pert = pert.to(device)
            time_emb = time_emb.to(device)
            x_target_delta = x_target_delta.to(device)

            outputs = model(x_baseline, pert, time_emb)
            x_target_abs = x_baseline + x_target_delta
            x_pred_abs = x_baseline + outputs['predicted_expr']

            all_predictions.append(x_pred_abs.cpu().numpy())
            all_targets.append(x_target_abs.cpu().numpy())
            all_baselines.append(x_baseline.cpu().numpy())
            all_perts.append(pert.cpu().numpy())

    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)
    all_baselines = np.concatenate(all_baselines, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)

    control_baseline = all_baselines
    results = calculate_detailed_metrics(
        all_predictions,
        all_targets,
        control_baseline=control_baseline,
        perturbations=all_perts
    )

    torch.save({
        'model_state_dict': model.state_dict(),
        'evaluation_results': results,
        'predictions': all_predictions,
        'targets': all_targets,
        'baselines': all_baselines,
        'gene_names': common_genes_info['genes'] if common_genes_info is not None else None,
        'pca_model': pca_model,
        'scaler': scaler,
        'best_params': best_params,
        'model_config': {
            'input_dim': model.input_dim,
            'pert_dim': model.pert_dim,
            'time_dim': model.time_dim,
            'latent_dim': model.latent_dim,
            'hidden_dim': model.hidden_dim,
            'model_in_dim': model.model_in_dim,
            'bottleneck_dims': model.bottleneck_dims,
            'use_bottleneck': model.use_bottleneck
        }
    }, save_path)

    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
```

```
                results['MSE_DE'], results['PCC_DE'], results['R2_DE']]
    })
    print("\nEvaluation Results:")
    print(metrics_df.to_string(index=False,
          float_format=lambda x: '{:.6f}'.format(x)))
    print(f"\nModel and evaluation results saved to: {save_path}")
    return results


def standardize_perturbation_encoding(*datasets):
    """
    ??? Dataset ?? one-hot ??????
    """
    if len(datasets) == 0:
        raise ValueError(
            "standardize_perturbation_encoding requires at least one dataset.")

    # ???? perturbation ??
    all_pert_names = set()
    for ds in datasets:
        all_pert_names.update(ds.perturbation_names)
    all_pert_names = sorted(list(all_pert_names))

    # ??? dataset ?? encode
    for ds in datasets:
        pert_df = pd.DataFrame(ds.adata.obs['perturbation'])
        pert_encoded = pd.get_dummies(pert_df['perturbation'])
        for name in all_pert_names:
            if name not in pert_encoded.columns:
                pert_encoded[name] = 0
        pert_encoded = pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        ds.perturbations = pert_encoded.values.astype(np.float32)
        ds._pert_encoding_standardized = True

    actual_pert_dim = len(all_pert_names)
    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names


def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, train_dataset, test_dataset, device, pca_model, scaler

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print_log(f'Training started at: {timestamp}')

    # ===== Device =====
    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print_log(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print_log(
                    f'Warning: Specified GPU {gpu_id} does not exist, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print_log(f'Using specified GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print_log(f'Using default GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print_log('CUDA not available, using CPU')

    # ===== Load data =====
    print_log('Loading data...')
    train_path = "datasets/SchiebingerLander2019_train_processed_filtered2.h5ad"
    test_path = "datasets/SchiebingerLander2019_test_processed_filtered2.h5ad"

    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(
            f"Data files not found: {train_path} or {test_path}")

    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print_log(f'Training data shape: {train_adata.shape}')
    print_log(f'Test data shape: {test_adata.shape}')

    # ===== Common genes =====
    print_log("Processing gene set consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print_log(f"Training genes: {len(train_genes)}")
    print_log(f"Test genes: {len(test_genes)}")
    print_log(f"Common genes: {len(common_genes)}")

    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(
        gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(
        gene) for gene in common_genes]

    pca_model = None
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]

    # ?????? fit scaler?log1p ????????? transform ? Dataset ???
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
```

```
train_data = np.log1p(train_data)
scaler = StandardScaler().fit(train_data)

common_genes_info = {
    'genes': common_genes,
    'train_idx': train_gene_idx,
    'test_idx': test_gene_idx
}

# ===== ?? train / val =====
n_train_cells = train_adata.n_obs
indices = np.arange(n_train_cells)
np.random.shuffle(indices)
train_size = int(0.8 * n_train_cells)
train_indices = indices[:train_size]
val_indices = indices[train_size:]

train_adata_train = train_adata[train_indices, :].copy()
train_adata_val = train_adata[val_indices, :].copy()

# ===== Build datasets =====
train_dataset = CytokineTrajectoryDataset(
    train_adata_train,
    perturbation_key='perturbation',
    age_key='age',
    scaler=scaler,
    pca_model=None,
    pca_dim=128,
    fit_pca=False,
    augment=True,
    is_train=True,
    common_genes_info=common_genes_info
)
val_dataset = CytokineTrajectoryDataset(
    train_adata_val,
    perturbation_key='perturbation',
    age_key='age',
    scaler=scaler,
    pca_model=None,
    pca_dim=128,
    fit_pca=False,
    augment=False,     # ?????????
    is_train=False,    # ???? baseline-target pairing?test ???
    common_genes_info=common_genes_info
)
test_dataset = CytokineTrajectoryDataset(
    test_adata,
    perturbation_key='perturbation',
    age_key='age',
    scaler=scaler,
    pca_model=None,
    pca_dim=128,
    fit_pca=False,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info
)

# ===== ?? perturbation ?? =====
pert_dim, _ = standardize_perturbation_encoding(
    train_dataset, val_dataset, test_dataset)

n_genes = train_dataset.n_genes
print_log(
    f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")

model = CytokineTrajectoryModel(
    input_dim=n_genes,
    output_dim=n_genes,
    pert_dim=pert_dim,
    time_dim=3,
    latent_dim=128,
    hidden_dim=512
).to(device)

# ===== DataLoaders =====
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
val_loader = DataLoader(
    val_dataset,
    batch_size=64,
    shuffle=False,
    num_workers=4,
    pin_memory=True
)
test_loader = DataLoader(
    test_dataset,
    batch_size=64,
    shuffle=False,
    num_workers=4,
    pin_memory=True
)

optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=5e-3,
    weight_decay=1e-4
)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer,
```

```
        T_0=10,
        T_mult=2,
        eta_min=1e-6
    )

    best_params = {
        'input_dim': n_genes,
        'output_dim': n_genes,
        'pert_dim': pert_dim,
        'time_dim': 3,
        'latent_dim': 128,
        'hidden_dim': 512,
        'learning_rate': 5e-3,
        'weight_decay': 1e-4,
        'batch_size': 64
    }

    print_log('Training model...')
    best_loss = float('inf')
    best_model_state = None
    max_epochs = 100
    patience = 15
    patience_counter = 0

    for epoch in range(max_epochs):
        train_loss = train_model(
            model, train_loader, optimizer, scheduler, device)
        val_metrics = evaluate_model(model, val_loader, device)

        if (epoch + 1) % 10 == 0:
            print_log(f'Epoch {epoch + 1}/{max_epochs}:')
            print_log(f'Training Loss: {train_loss:.4f}')
            print_log(f'Validation Loss: {val_metrics["loss"]:.4f}')

        if val_metrics["loss"] < best_loss:
            best_loss = val_metrics["loss"]
            patience_counter = 0
            best_model_state = copy.deepcopy(model.state_dict())
            torch.save({
                'epoch': epoch,
                'model_state_dict': best_model_state,
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss': best_loss,
                'metrics': val_metrics,
                'best_params': best_params
            }, f'cytokines_best_model_{timestamp}.pt')
            print_log(
                f"Saved best model with validation loss: {best_loss:.4f}")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print_log(f'Early stopping at epoch {epoch + 1}')
                break

    if best_model_state is not None:
        model.load_state_dict(best_model_state)

    print_log('Evaluating final model on test set...')
    results = evaluate_and_save_model(
        model,
        test_loader,
        device,
        f'cytokines_final_model_{timestamp}.pt',
        common_genes_info,
        pca_model,
        scaler,
        best_params
    )

    best_params_str = str(best_params)
    if len(best_params_str) > 50:
        best_params_str = best_params_str[:50] + '...'

    results_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']],
        'Best_Params': [best_params_str] * 6
    })
    results_df.to_csv(
        f'cytokines_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(
        index=False, float_format=lambda x: '{:.6f}'.format(x)))
    return results_df


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Cytokine Trajectory Model Training')
    parser.add_argument('--gpu', type=int, default=None,
                        help='Specify GPU ID to use (e.g., --gpu 0 for GPU 0, --gpu 1 for GPU 1)')
    parser.add_argument('--list-gpus', action='store_true',
                        help='List all available GPUs and exit')
    args = parser.parse_args()

    if args.list_gpus:
        if torch.cuda.is_available():
            gpu_count = torch.cuda.device_count()
            print(f'Available GPUs: {gpu_count}')
            for i in range(gpu_count):
                print(f'GPU {i}: {torch.cuda.get_device_name(i)}')
        else:
            print('CUDA not available')
```

```
        exit(0)

    print("=" * 60)
    print("Cytokine Trajectory Model Training")
    print("=" * 60)

    if args.gpu is not None:
        print(f"Using specified GPU: {args.gpu}")
    else:
        print("Using default GPU settings")

    results_df = main(gpu_id=args.gpu)
```

D.5 Papalexi CITE-seq RNA-readout implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
from IPython.display import display
import time
from datetime import datetime
import argparse
import warnings
import random
warnings.filterwarnings('ignore')
def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")

def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
pca_model = None
scaler = None
class CITERNADataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', scaler=None, pca_model=None,
                 pca_dim=128, fit_pca=False, augment=False, is_train=True, common_genes_info=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.augment = augment
        self.training = True
        self.pca_dim = pca_dim
        self.is_train = is_train
        self.common_genes_info = common_genes_info
        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X
        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)
        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)
        if not is_log1p_already:
            data = np.log1p(data)
        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
```

```
            if not is_standardized_already:
                data = self.scaler.transform(data)
        data = np.clip(data, -10, 10)
        data = data / 10.0
        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self.pca = pca_model if pca_model is not None else None
        self.pca_dim = pca_dim if pca_model is not None else None
        self.perturbation_names = list(adata.obs[perturbation_key].unique())
        self.perturbations = pd.get_dummies(adata.obs[perturbation_key]).values
        print_log(
            f"{'train' if is_train else 'test'} set perturbation dimension: {self.perturbations.shape[1]}")
        self._create_baseline_perturbation_pairs()
    def _create_baseline_perturbation_pairs(self):
        perturbation_col = self.adata.obs[self.perturbation_key].astype(str)
        negctrl_mask = perturbation_col.str.contains('NegCtrl', case=False, na=False)
        lower_labels = perturbation_col.str.lower()
        exact_control_mask = (lower_labels == 'control')
        other_control_mask = perturbation_col.isin(['Control', 'ctrl', 'Ctrl', 'CONTROL'])
        contains_control_mask = perturbation_col.str.contains('control|ctrl', case=False, na=False)
        control_mask = exact_control_mask | negctrl_mask | other_control_mask | contains_control_mask
        control_indices = np.where(control_mask)[0]
        perturbed_mask = ~control_mask
        perturbed_indices = np.where(perturbed_mask)[0]
        print_log(f"Control samples: {len(control_indices)}")
        print_log(f"Perturbed samples: {len(perturbed_indices)}")
        if len(control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
                    "This model performs control -> perturbed prediction, NOT autoencoder. "
                    "Please ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                raise ValueError(
                    "CRITICAL ERROR: Test set has NO control samples! "
                    "For unseen perturbation prediction, test set should use training set control baseline. "
                    "Please ensure training set has control samples."
                )
        self.pairs = []
        for i, perturbed_idx in enumerate(perturbed_indices):
            if self.is_train:
                baseline_idx = np.random.choice(control_indices)
            else:
                baseline_idx = control_indices[i % len(control_indices)]
            self.pairs.append({
                'baseline_idx': baseline_idx,
                'perturbed_idx': perturbed_idx,
                'perturbation': self.perturbations[perturbed_idx]
            })
        for i, control_idx in enumerate(control_indices):
            if len(perturbed_indices) > 0:
                if self.is_train:
                    target_idx = np.random.choice(perturbed_indices)
                else:
                    target_idx = perturbed_indices[i % len(perturbed_indices)]
                self.pairs.append({
                    'baseline_idx': control_idx,
                    'perturbed_idx': target_idx,
                    'perturbation': self.perturbations[target_idx]
                })
            else:
                alternative_controls = control_indices[control_indices != control_idx]
                if len(alternative_controls) > 0:
                    if self.is_train:
                        target_idx = np.random.choice(alternative_controls)
                    else:
                        target_idx = alternative_controls[i % len(alternative_controls)]
                    self.pairs.append({
                        'baseline_idx': control_idx,
                        'perturbed_idx': target_idx,
                        'perturbation': self.perturbations[target_idx]
                    })
        print_log(f"Total pairs created: {len(self.pairs)}")
    def __len__(self):
        return len(self.pairs)
    def __getitem__(self, idx):
        pair = self.pairs[idx]
        baseline_expr = self.expression_data[pair['baseline_idx']]
        perturbed_expr = self.original_expression_data[pair['perturbed_idx']]
        perturbation = pair['perturbation']
        # Target is now perturbed expression directly (not delta)
        target_expr = perturbed_expr
        if self.augment and self.training:
            noise = np.random.normal(0, 0.05, baseline_expr.shape)
            baseline_expr = baseline_expr + noise
            mask = np.random.random(baseline_expr.shape) > 0.05
            baseline_expr = baseline_expr * mask
            scale = np.random.uniform(0.95, 1.05)
            baseline_expr = baseline_expr * scale
        return (torch.FloatTensor(baseline_expr),
                torch.FloatTensor(perturbation),
                torch.FloatTensor(target_expr))
class PerturbationEmbedding(nn.Module):
    def __init__(self, pert_dim, emb_dim):
        super().__init__()
        self.embedding = nn.Linear(pert_dim, emb_dim)
        self.norm = nn.LayerNorm(emb_dim)
        self.dropout = nn.Dropout(0.1)
    def forward(self, pert):
        x = self.embedding(pert)
        x = self.norm(x)
        x = F.gelu(x)
```

```
        x = self.dropout(x)
        return x
class CITERNATransformerModel(nn.Module):
    def __init__(self, input_dim, output_dim, pert_dim, hidden_dim=512, n_layers=3, n_heads=8,
                 dropout=0.1, attention_dropout=0.1, ffn_dropout=0.1,
                 use_pert_emb=True, pert_emb_dim=64,
                 model_in_dim=128, bottleneck_dims=(2048, 512), use_bottleneck=True):
        super(CITERNATransformerModel, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.model_in_dim = model_in_dim
        self.pert_dim = pert_dim
        self.hidden_dim = hidden_dim
        self.use_pert_emb = use_pert_emb
        self.use_bottleneck = use_bottleneck
        if self.use_bottleneck:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, bottleneck_dims[0]),
                nn.LayerNorm(bottleneck_dims[0]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[0], bottleneck_dims[1]),
                nn.LayerNorm(bottleneck_dims[1]),
                nn.GELU(),
                nn.Dropout(0.1),
                nn.Linear(bottleneck_dims[1], self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
            )
        else:
            self.input_proj = nn.Sequential(
                nn.Linear(self.input_dim, self.model_in_dim),
                nn.LayerNorm(self.model_in_dim),
                nn.GELU()
            )
        self.expression_encoder = nn.Sequential(
            nn.Linear(self.model_in_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        if use_pert_emb:
            self.pert_encoder = PerturbationEmbedding(pert_dim, pert_emb_dim)
            pert_out_dim = pert_emb_dim
        else:
            self.pert_encoder = nn.Sequential(
                nn.Linear(pert_dim, hidden_dim),
                nn.LayerNorm(hidden_dim),
                nn.GELU(),
                nn.Dropout(dropout)
            )
            pert_out_dim = hidden_dim
        fusion_dim = hidden_dim + pert_out_dim
        self.fusion_dim = ((fusion_dim + n_heads - 1) // n_heads) * n_heads
        if self.fusion_dim != fusion_dim:
            self.fusion_proj = nn.Linear(fusion_dim, self.fusion_dim)
        else:
            self.fusion_proj = nn.Identity()

        # Split fusion features into multiple tokens for meaningful attention
        self.n_tokens = 8
        self.token_dim = ((self.fusion_dim // self.n_tokens + n_heads - 1) // n_heads) * n_heads
        self.token_proj = nn.Linear(self.fusion_dim, self.n_tokens * self.token_dim)

        # Learnable positional embeddings
        self.pos_embedding = nn.Parameter(torch.randn(1, self.n_tokens, self.token_dim) * 0.02)

        # Real transformer encoder layers
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=self.token_dim,
            nhead=n_heads,
            dim_feedforward=self.token_dim * 4,
            dropout=ffn_dropout,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=n_layers)

        # Project back from tokens to fusion dim
        self.token_unproj = nn.Linear(self.n_tokens * self.token_dim, self.fusion_dim)

        self.fusion = nn.Sequential(
            nn.Linear(self.fusion_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.output = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, output_dim)
        )
        self.perturbation_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim//2),
            nn.LayerNorm(hidden_dim//2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim//2, pert_dim)
        )
```

```
        self.apply(self._init_weights)
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)
    def forward(self, baseline_expr, pert):
        assert baseline_expr.dim() == 2 and baseline_expr.size(1) == self.input_dim, \
            f"Expected baseline_expr shape [B, {self.input_dim}], got {baseline_expr.shape}"
        batch_size = baseline_expr.size(0)
        baseline_expr_proj = self.input_proj(baseline_expr)
        expr_feat = self.expression_encoder(baseline_expr_proj)
        pert_feat = self.pert_encoder(pert)
        fusion_input = torch.cat([expr_feat, pert_feat], dim=1)
        fusion_input = self.fusion_proj(fusion_input)

        # Project and reshape into tokens: [batch, fusion_dim] -> [batch, n_tokens, token_dim]
        tokens = self.token_proj(fusion_input)
        tokens = tokens.view(batch_size, self.n_tokens, self.token_dim)

        # Add positional embeddings
        tokens = tokens + self.pos_embedding

        # Apply transformer encoder
        x_trans = self.transformer(tokens)

        # Flatten tokens and project back: [batch, n_tokens, token_dim] -> [batch, fusion_dim]
        x_trans = x_trans.view(batch_size, -1)
        x_trans = self.token_unproj(x_trans)

        fused = self.fusion(x_trans)
        output = self.output(fused)
        pert_pred = self.perturbation_head(fused)
        return output, pert_pred
def train_model(model, train_loader, optimizer, scheduler, device, aux_weight=0.1):
    model.train()
    total_loss = 0
    accumulation_steps = 4
    optimizer.zero_grad()
    for i, batch in enumerate(train_loader):
        baseline_expr, pert, target_expr = batch
        baseline_expr, pert, target_expr = baseline_expr.to(
            device), pert.to(device), target_expr.to(device)
        output, pert_pred = model(baseline_expr, pert)
        main_loss = F.mse_loss(output, target_expr)
        aux_loss = F.mse_loss(pert_pred, pert)
        loss = main_loss + aux_weight * aux_loss
        loss = loss / accumulation_steps
        loss.backward()
        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()
        total_loss += loss.item() * accumulation_steps
    return total_loss / len(train_loader)
def evaluate_model(model, test_loader, device, aux_weight=0.1):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    all_perts = []
    all_pert_preds = []
    with torch.no_grad():
        for batch in test_loader:
            baseline_expr, pert, target_expr = batch
            baseline_expr, pert, target_expr = baseline_expr.to(
                device), pert.to(device), target_expr.to(device)
            output, pert_pred = model(baseline_expr, pert)
            main_loss = F.mse_loss(output, target_expr)
            aux_loss = F.mse_loss(pert_pred, pert)
            loss = main_loss + aux_weight * aux_loss
            total_loss += loss.item()
            all_targets.append(target_expr.cpu().numpy())
            all_predictions.append(output.cpu().numpy())
            all_perts.append(pert.cpu().numpy())
            all_pert_preds.append(pert_pred.cpu().numpy())
    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)
    all_pert_preds = np.concatenate(all_pert_preds, axis=0)
    r2 = r2_score(all_targets, all_predictions)
    if np.isnan(r2):
        r2 = 0.0
    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pearson = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pearson):
            pearson = 0.0
    else:
        pearson = 0.0
    pert_r2 = r2_score(all_perts, all_pert_preds)
    if np.isnan(pert_r2):
        pert_r2 = 0.0
    return {
        'loss': total_loss / len(test_loader),
        'r2': r2,
        'pearson': pearson,
        'pert_r2': pert_r2
    }
def calculate_metrics(pred, true, control_baseline=None, perturbations=None):
```

```
    n_samples, n_genes = true.shape
    mse = np.mean((pred - true) ** 2)
    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0
    true_mean_overall = np.mean(true)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_overall) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0
    if control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) / (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
                    pred_de_flat = pred_de.flatten()
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_overall = np.mean(true_de)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_overall) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            mse_de = np.mean((pred[de_mask] - true[de_mask]) ** 2)
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                pred_de_flat = pred_de.flatten()
                true_de_flat = true_de.flatten()
                if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                    pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0
            else:
                pcc_de = 0.0
            true_de_mean = np.mean(true[de_mask])
            ss_res_de = np.sum((pred[de_mask] - true[de_mask]) ** 2)
            ss_tot_de = np.sum((true[de_mask] - true_de_mean) ** 2)
            if ss_tot_de > 1e-10:
                r2_de = 1.0 - (ss_res_de / ss_tot_de)
            else:
                r2_de = 0.0
            if np.isnan(r2_de):
                r2_de = 0.0
        else:
            mse_de = pcc_de = r2_de = np.nan
    return {
        'MSE': mse,
```

```
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }
def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(list(set(train_dataset.perturbation_names + test_dataset.perturbation_names)))
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []
    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))
    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(
        columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)
    for i, pair in enumerate(train_dataset.pairs):
        pert_idx = pair['perturbed_idx']
        train_dataset.pairs[i]['perturbation'] = train_dataset.perturbations[pert_idx]
    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(np.float32)
        for i, pair in enumerate(test_dataset.pairs):
            pert_idx = pair['perturbed_idx']
            test_dataset.pairs[i]['perturbation'] = test_dataset.perturbations[pert_idx]
        test_dataset._pert_encoding_standardized = True
    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True
    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names
def objective(trial, timestamp):
    global train_dataset, test_dataset, device, pca_model, test_adata
    params = {
        'pca_dim': 128,
        'n_hidden': trial.suggest_int('n_hidden', 256, 1024, step=8),
        'n_layers': trial.suggest_int('n_layers', 2, 4),
        'n_heads': trial.suggest_categorical('n_heads', [4, 8]),
        'dropout': trial.suggest_float('dropout', 0.1, 0.3),
        'attention_dropout': trial.suggest_float('attention_dropout', 0.1, 0.2),
        'ffn_dropout': trial.suggest_float('ffn_dropout', 0.1, 0.2),
        'aux_weight': trial.suggest_float('aux_weight', 0.05, 0.15),
        'learning_rate': trial.suggest_float('learning_rate', 1e-4, 1e-3, log=True),
        'weight_decay': trial.suggest_float('weight_decay', 1e-5, 1e-4, log=True),
        'batch_size': trial.suggest_categorical('batch_size', [32, 64, 128]),
        'use_pert_emb': trial.suggest_categorical('use_pert_emb', [True, False]),
        'pert_emb_dim': trial.suggest_int('pert_emb_dim', 32, 128)
    }
    test_perturbation_names = list(test_adata.obs['perturbation'].unique())
    pert_dim, _ = standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=test_perturbation_names)
    n_genes = train_dataset.n_genes
    print(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
    model = CITERNATransformerModel(
        input_dim=n_genes,
        output_dim=n_genes,
        pert_dim=pert_dim,
        hidden_dim=params['n_hidden'],
        n_layers=params['n_layers'],
        n_heads=params['n_heads'],
        dropout=params['dropout'],
        attention_dropout=params['attention_dropout'],
        ffn_dropout=params['ffn_dropout'],
        use_pert_emb=params['use_pert_emb'],
        pert_emb_dim=params['pert_emb_dim']
    ).to(device)
    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=params['learning_rate'],
        weight_decay=params['weight_decay']
    )
    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer, T_0=10, T_mult=2, eta_min=1e-6
    )
    train_size = int(0.8 * len(train_dataset))
    val_size = len(train_dataset) - train_size
    train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
    train_loader = DataLoader(
        train_subset,
        batch_size=params['batch_size'],
        shuffle=True,
        num_workers=4,
```

```
        pin_memory=True
    )
    val_loader = DataLoader(
        val_subset,
        batch_size=params['batch_size'],
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
    best_val_loss = float('inf')
    patience = 15
    patience_counter = 0
    max_epochs = 150
    for epoch in range(max_epochs):
        model.train()
        train_loss = 0
        for batch in tqdm(train_loader, desc=f'Epoch {epoch+1}/{max_epochs}'):
            baseline_expr, pert, target_expr = batch
            baseline_expr, pert, target_expr = baseline_expr.to(
                device), pert.to(device), target_expr.to(device)
            optimizer.zero_grad()
            output, pert_pred = model(baseline_expr, pert)
            mse_loss = F.mse_loss(output, target_expr)
            pert_loss = F.mse_loss(pert_pred, pert)
            loss = mse_loss + params['aux_weight'] * pert_loss
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            train_loss += loss.item()
        train_loss /= len(train_loader)
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for batch in val_loader:
                baseline_expr, pert, target_expr = batch
                baseline_expr, pert, target_expr = baseline_expr.to(
                    device), pert.to(device), target_expr.to(device)
                output, pert_pred = model(baseline_expr, pert)
                mse_loss = F.mse_loss(output, target_expr)
                pert_loss = F.mse_loss(pert_pred, pert)
                loss = mse_loss + params['aux_weight'] * pert_loss
                val_loss += loss.item()
        val_loss /= len(val_loader)
        scheduler.step()
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
            torch.save(model.state_dict(),
                       f'best_rna_model_trial_{trial.number}_{timestamp}.pt')
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f'Early stopping at epoch {epoch+1}')
                break
        if (epoch + 1) % 20 == 0:
            print(
                f'Epoch {epoch+1}/{max_epochs} - Train Loss: {train_loss:.4f} - Val Loss: {val_loss:.4f}')
    return best_val_loss
def evaluate_and_save_model(model, test_loader, device, save_path, common_genes_info=None, pca_model=None, scaler=None, best_params=None, all_pert_names=None):
    model.eval()
    all_predictions = []
    all_targets = []
    all_baselines = []
    all_perts = []
    with torch.no_grad():
        for batch in tqdm(test_loader, desc='Evaluating'):
            baseline_expr, pert, target_expr = batch
            baseline_expr, pert, target_expr = baseline_expr.to(
                device), pert.to(device), target_expr.to(device)
            output, _ = model(baseline_expr, pert)
            all_predictions.append(output.cpu().numpy())
            all_targets.append(target_expr.cpu().numpy())
            all_baselines.append(baseline_expr.cpu().numpy())
            all_perts.append(pert.cpu().numpy())
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)
    all_baselines = np.concatenate(all_baselines, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)
    control_baseline = all_baselines
    results = calculate_metrics(all_predictions, all_targets, control_baseline=control_baseline, perturbations=all_perts)
    torch.save({
        'model_state_dict': model.state_dict(),
        'evaluation_results': results,
        'predictions': all_predictions,
        'targets': all_targets,
        'baselines': all_baselines,
        'gene_names': common_genes_info['genes'] if common_genes_info is not None else None,
        'common_genes_info': common_genes_info,
        'pca_model': pca_model,
        'scaler': scaler,
        'best_params': best_params,
        'model_config': {
            'input_dim': model.input_dim,
            'pert_dim': model.pert_dim,
            'hidden_dim': getattr(model, 'hidden_dim', 512),
            'n_layers': getattr(model, 'n_layers', 3),
            'n_heads': getattr(model, 'n_heads', 8),
            'dropout': getattr(model, 'dropout', 0.1),
            'use_pert_emb': getattr(model, 'use_pert_emb', True)
        },
        'perturbation_encoding': {
            'pert_dim': model.pert_dim,
            'all_pert_names': all_pert_names
        }
    }, save_path)
```

```
    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']]
    })
    print_log("\nRNA Model Evaluation Results:")
    print_log(metrics_df.to_string(index=False,
          float_format=lambda x: '{:.6f}'.format(x)))
    print_log(f"\nModel and evaluation results saved to: {save_path}")
    return results
def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, pca_model, scaler
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print(f'RNA Model Training started at: {timestamp}')
    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print(f'Warning: GPU {gpu_id} not available, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print(f'Using GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print(f'Using default GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print('CUDA not available, using CPU')
    print('Loading CITE-seq RNA data...')
    train_path = "./datasets/PapalexiSatija2021_train.h5ad"
    test_path = "./datasets/PapalexiSatija2021_test.h5ad"
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(
            f"Data files not found: {train_path} or {test_path}")
    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print(f'Training data shape: {train_adata.shape}')
    print(f'Test data shape: {test_adata.shape}')
    print("Processing gene consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print(f"Training genes: {len(train_genes)}")
    print(f"Test genes: {len(test_genes)}")
    print(f"Common genes: {len(common_genes)}")
    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(
        gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(
        gene) for gene in common_genes]
    pca_model = None
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]
    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    scaler = StandardScaler()
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0
    common_genes_info = {
        'genes': common_genes,
        'train_idx': train_gene_idx,
        'test_idx': test_gene_idx
    }
    train_dataset = CITERNADataset(
        train_adata,
        perturbation_key='perturbation',
        scaler=scaler,
        pca_model=None,
        pca_dim=128,
        fit_pca=False,
        augment=True,
        is_train=True,
        common_genes_info=common_genes_info
    )
    test_dataset = CITERNADataset(
        test_adata,
        perturbation_key='perturbation',
        scaler=scaler,
        pca_model=None,
        pca_dim=128,
        fit_pca=False,
        augment=False,
        is_train=False,
        common_genes_info=common_genes_info
    )
    pert_dim, all_pert_names = standardize_perturbation_encoding(train_dataset, test_dataset)
    optuna.logging.set_verbosity(optuna.logging.WARNING)
    study = optuna.create_study(
        direction='minimize',
        sampler=optuna.samplers.TPESampler(seed=42),
        pruner=optuna.pruners.MedianPruner(
            n_startup_trials=5,
            n_warmup_steps=5,
            interval_steps=1
        )
    )
    print('Starting hyperparameter optimization...')
    study.optimize(lambda trial: objective(trial, timestamp), n_trials=30)
    best_params = study.best_params
```

```
n_genes = train_dataset.n_genes
print(f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")
final_model = CITERNATransformerModel(
    input_dim=n_genes,
    output_dim=n_genes,
    pert_dim=pert_dim,
    hidden_dim=best_params['n_hidden'],
    n_layers=best_params['n_layers'],
    n_heads=best_params['n_heads'],
    dropout=best_params['dropout'],
    attention_dropout=best_params['attention_dropout'],
    ffn_dropout=best_params['ffn_dropout'],
    use_pert_emb=best_params['use_pert_emb'],
    pert_emb_dim=best_params['pert_emb_dim']
).to(device)
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(train_dataset, [train_size, val_size])
train_loader = DataLoader(
    train_subset,
    batch_size=best_params['batch_size'],
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
val_loader = DataLoader(
    val_subset,
    batch_size=best_params['batch_size'],
    shuffle=False,
    num_workers=4,
    pin_memory=True
)
test_loader = DataLoader(
    test_dataset,
    batch_size=best_params['batch_size'],
    shuffle=False,
    num_workers=4,
    pin_memory=True
)
optimizer = torch.optim.AdamW(
    final_model.parameters(),
    lr=best_params['learning_rate'],
    weight_decay=best_params['weight_decay']
)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer, T_0=10, T_mult=2, eta_min=1e-6
)
print('Training final RNA model...')
best_loss = float('inf')
best_model = None
max_epochs = 200
patience = 20
patience_counter = 0
for epoch in range(max_epochs):
    train_loss = train_model(final_model, train_loader, optimizer, scheduler, device,
                             aux_weight=best_params['aux_weight'])
    val_metrics = evaluate_model(final_model, val_loader, device,
                                 aux_weight=best_params['aux_weight'])
    if (epoch + 1) % 20 == 0:
        print(f'Epoch {epoch+1}/{max_epochs}:')
        print(f'Training Loss: {train_loss:.4f}')
        print(f'Validation Loss: {val_metrics["loss"]:.4f}')
    if val_metrics["loss"] < best_loss:
        best_loss = val_metrics["loss"]
        patience_counter = 0
        best_model = final_model.state_dict()
        torch.save({
            'epoch': epoch,
            'model_state_dict': best_model,
            'optimizer_state_dict': optimizer.state_dict(),
            'scheduler_state_dict': scheduler.state_dict(),
            'loss': best_loss,
            'metrics': val_metrics,
            'best_params': best_params,
            'scaler': scaler,
            'common_genes_info': common_genes_info,
            'perturbation_encoding': {
                'pert_dim': pert_dim,
                'all_pert_names': all_pert_names
            }
        }, f'cite_rna_best_model_{timestamp}.pt')
        print(f"Saved best RNA model with validation loss: {best_loss:.4f}")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print(f'Early stopping at epoch {epoch+1}')
            break
final_model.load_state_dict(best_model)
print_log('Evaluating final RNA model on test set...')
results = evaluate_and_save_model(final_model, test_loader, device,
                                  f'cite_rna_final_model_{timestamp}.pt',
                                  common_genes_info, pca_model, scaler, best_params, all_pert_names)

best_params_str = str(best_params)
if len(best_params_str) > 50:
    best_params_str = best_params_str[:50] + '...'
results_df = pd.DataFrame({
    'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
    'Value': [results['MSE'], results['PCC'], results['R2'],
              results['MSE_DE'], results['PCC_DE'], results['R2_DE']],
    'Best_Params': [best_params_str] * 6
})
results_df.to_csv(
    f'cite_rna_evaluation_results_{timestamp}.csv', index=False)
print_log("\nFinal Evaluation Results:")
```

```
    print_log(results_df.to_string(index=False, float_format=lambda x: '{:.6f}'.format(x)))
    return results_df
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='CITE-seq RNA Model Training')
    parser.add_argument('--gpu', type=int, default=None,
                        help='Specify GPU ID (0-7)')
    parser.add_argument('--list-gpus', action='store_true',
                        help='List available GPUs and exit')
    args = parser.parse_args()
    if args.list_gpus:
        if torch.cuda.is_available():
            gpu_count = torch.cuda.device_count()
            print(f'Available GPUs: {gpu_count}')
            for i in range(gpu_count):
                print(f'GPU {i}: {torch.cuda.get_device_name(i)}')
        else:
            print('CUDA not available')
        exit(0)
    print("=" * 60)
    print("CITE-seq RNA Model Training")
    print("=" * 60)
    if args.gpu is not None:
        print(f"Using GPU: {args.gpu}")
    else:
        print("Using default GPU settings")
    results_df = main(gpu_id=args.gpu)
```

D.6 *totalGAT* for Papalexi CITE-seq (protein readout)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import RobustScaler
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
import time
from datetime import datetime
import argparse
import warnings
import random
warnings.filterwarnings('ignore')

def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")

def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
scaler = None

class CITEProteinDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', scaler=None,
                 augment=False, is_train=True, common_genes_info=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.augment = augment
        self.training = True
        self.is_train = is_train
        self.common_genes_info = common_genes_info

        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']

            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X

        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data = np.arcsinh(data / 5.0)

        if scaler is None:
            self.scaler = RobustScaler()
```

```
            data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            data = self.scaler.transform(data)

        data = np.clip(data, -5, 5)

        self.protein_data = data

        perturbation_col = self.adata.obs[perturbation_key].astype(str)
        negctrl_mask = perturbation_col.str.contains('NegCtrl', case=False, na=False)
        lower_labels = perturbation_col.str.lower()
        exact_control_mask = (lower_labels == 'control')
        other_control_mask = perturbation_col.isin(['Control', 'ctrl', 'Ctrl', 'CONTROL'])
        contains_control_mask = perturbation_col.str.contains('control|ctrl', case=False, na=False)
        control_mask = exact_control_mask | negctrl_mask | other_control_mask | contains_control_mask

        self.perturbation_names = list(perturbation_col.unique())
        self.perturbations = pd.get_dummies(perturbation_col).values.astype(np.float32)
        print_log(f"{'train' if is_train else 'test'} set perturbation dimension: {self.perturbations.shape[1]}")
        print_log(f"{'train' if is_train else 'test'} set protein dimension: {self.protein_data.shape[1]}")

        self._create_baseline_perturbation_pairs()

    def _create_baseline_perturbation_pairs(self):
        perturbation_col = self.adata.obs[self.perturbation_key].astype(str)
        negctrl_mask = perturbation_col.str.contains('NegCtrl', case=False, na=False)
        lower_labels = perturbation_col.str.lower()
        exact_control_mask = (lower_labels == 'control')
        other_control_mask = perturbation_col.isin(['Control', 'ctrl', 'Ctrl', 'CONTROL'])
        contains_control_mask = perturbation_col.str.contains('control|ctrl', case=False, na=False)
        control_mask = exact_control_mask | negctrl_mask | other_control_mask | contains_control_mask

        control_indices = np.where(control_mask)[0]
        perturbed_mask = ~control_mask
        perturbed_indices = np.where(perturbed_mask)[0]

        print_log(f"Control samples: {len(control_indices)}")
        print_log(f"Perturbed samples: {len(perturbed_indices)}")

        if len(control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
                    "This model performs control -> perturbed prediction, NOT autoencoder. "
                    "Please ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                raise ValueError(
                    "CRITICAL ERROR: Test set has NO control samples! "
                    "For unseen perturbation prediction, test set should use training set control baseline. "
                    "Please ensure training set has control samples."
                )

        self.pairs = []
        for i, perturbed_idx in enumerate(perturbed_indices):
            if self.is_train:
                baseline_idx = np.random.choice(control_indices)
            else:
                baseline_idx = control_indices[i % len(control_indices)]
            self.pairs.append({
                'baseline_idx': baseline_idx,
                'perturbed_idx': perturbed_idx,
                'perturbation': self.perturbations[perturbed_idx]
            })

        for i, control_idx in enumerate(control_indices):
            if len(perturbed_indices) > 0:
                if self.is_train:
                    target_idx = np.random.choice(perturbed_indices)
                else:
                    target_idx = perturbed_indices[i % len(perturbed_indices)]
                self.pairs.append({
                    'baseline_idx': control_idx,
                    'perturbed_idx': target_idx,
                    'perturbation': self.perturbations[target_idx]
                })

        print_log(f"Total pairs created: {len(self.pairs)}")

    def __len__(self):
        return len(self.pairs)

    def __getitem__(self, idx):
        pair = self.pairs[idx]
        baseline_idx = pair['baseline_idx']
        target_idx = pair['perturbed_idx']

        baseline_protein = self.protein_data[baseline_idx]
        target_protein = self.protein_data[target_idx]
        perturbation = pair['perturbation']

        if self.augment and self.training:
            noise = np.random.normal(0, 0.01, baseline_protein.shape)
            baseline_protein = baseline_protein + noise
            mask = np.random.random(baseline_protein.shape) > 0.02
            baseline_protein = baseline_protein * mask
            scale = np.random.uniform(0.99, 1.01)
            baseline_protein = baseline_protein * scale

        # Return perturbed expression directly as target (not delta)
        return (torch.FloatTensor(baseline_protein),
                torch.FloatTensor(perturbation),
                torch.FloatTensor(target_protein))
```

```
class GraphAttentionLayer(nn.Module):
    def __init__(self, in_dim, out_dim, n_heads=4, dropout=0.1):
        super().__init__()
        self.n_heads = n_heads
        self.out_dim = out_dim
        # Ensure internal dimension is divisible by n_heads
        self.head_dim = (out_dim + n_heads - 1) // n_heads
        self.internal_dim = self.head_dim * n_heads

        self.W = nn.Linear(in_dim, self.internal_dim, bias=False)
        self.dropout = nn.Dropout(dropout)
        self.leaky_relu = nn.LeakyReLU(0.2)
        # Project back to out_dim if needed
        if self.internal_dim != out_dim:
            self.out_proj = nn.Linear(self.internal_dim, out_dim)
        else:
            self.out_proj = nn.Identity()

    def forward(self, h):
        batch_size, n_nodes, _ = h.shape

        Wh = self.W(h)
        Wh = Wh.view(batch_size, n_nodes, self.n_heads, self.head_dim)
        Wh = Wh.transpose(1, 2)  # [batch, heads, n_nodes, head_dim]

        # Self-attention: Q @ K^T / sqrt(head_dim)
        scale = self.head_dim ** -0.5
        e = torch.matmul(Wh, Wh.transpose(-2, -1)) * scale  # [batch, heads, n_nodes, n_nodes]

        attention = F.softmax(self.leaky_relu(e), dim=-1)
        attention = self.dropout(attention)

        h_prime = torch.matmul(attention, Wh)  # [batch, heads, n_nodes, head_dim]
        h_prime = h_prime.transpose(1, 2).contiguous()
        h_prime = h_prime.view(batch_size, n_nodes, self.internal_dim)
        h_prime = self.out_proj(h_prime)

        return h_prime

class CrossAttentionModule(nn.Module):
    def __init__(self, protein_dim, pert_dim, hidden_dim, n_heads=4, dropout=0.1):
        super().__init__()
        self.protein_dim = protein_dim
        self.pert_dim = pert_dim
        self.hidden_dim = hidden_dim

        # Project to Q, K, V separately for proper cross-attention
        self.protein_to_q = nn.Linear(protein_dim, hidden_dim)
        self.pert_to_kv = nn.Linear(pert_dim, 2 * hidden_dim)

        self.pert_to_q = nn.Linear(pert_dim, hidden_dim)
        self.protein_to_kv = nn.Linear(protein_dim, 2 * hidden_dim)

        self.multihead_attn1 = nn.MultiheadAttention(hidden_dim, n_heads, dropout=dropout, batch_first=True)
        self.multihead_attn2 = nn.MultiheadAttention(hidden_dim, n_heads, dropout=dropout, batch_first=True)

        self.fusion = nn.Sequential(
            nn.Linear(2 * hidden_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, hidden_dim)
        )

    def forward(self, protein_feat, pert_feat):
        protein_seq = protein_feat.unsqueeze(1)
        pert_seq = pert_feat.unsqueeze(1)

        # Cross-attention 1: protein attends to perturbation
        q1 = self.protein_to_q(protein_seq)
        k1, v1 = self.pert_to_kv(pert_seq).chunk(2, dim=-1)
        out1, _ = self.multihead_attn1(q1, k1, v1)

        # Cross-attention 2: perturbation attends to protein
        q2 = self.pert_to_q(pert_seq)
        k2, v2 = self.protein_to_kv(protein_seq).chunk(2, dim=-1)
        out2, _ = self.multihead_attn2(q2, k2, v2)

        combined = torch.cat([out1.squeeze(1), out2.squeeze(1)], dim=-1)
        fused = self.fusion(combined)

        return fused

class ProteinResidualBlock(nn.Module):
    def __init__(self, dim, dropout=0.1):
        super().__init__()
        self.linear1 = nn.Linear(dim, dim)
        self.linear2 = nn.Linear(dim, dim)
        self.norm1 = nn.LayerNorm(dim)
        self.norm2 = nn.LayerNorm(dim)
        self.dropout = nn.Dropout(dropout)
        self.activation = nn.GELU()

    def forward(self, x):
        residual = x
        x = self.norm1(x)
        x = self.activation(x)
        x = self.linear1(x)
        x = self.dropout(x)
        x = self.norm2(x)
        x = self.activation(x)
        x = self.linear2(x)
        return x + residual
```

```
class EnhancedProteinModel(nn.Module):
    def __init__(self, input_dim, pert_dim, hidden_dim=256, n_layers=3, n_heads=4,
                 dropout=0.1, attention_dropout=0.1, ffn_dropout=0.1,
                 use_gat=True, gat_layers=2, use_cross_attn=True):
        super(EnhancedProteinModel, self).__init__()
        self.input_dim = input_dim
        self.pert_dim = pert_dim
        self.hidden_dim = hidden_dim
        self.use_gat = use_gat
        self.use_cross_attn = use_cross_attn

        # Number of tokens for GAT and transformer
        self.n_gat_nodes = 8  # Split protein features into multiple nodes for GAT
        self.n_tokens = 8     # Number of tokens for transformer

        self.protein_encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )

        self.protein_residuals = nn.ModuleList([
            ProteinResidualBlock(hidden_dim, dropout) for _ in range(2)
        ])

        if use_gat:
            # Project to multiple nodes for meaningful graph attention
            self.gat_node_proj = nn.Linear(hidden_dim, self.n_gat_nodes * hidden_dim)
            self.gat_layers = nn.ModuleList()
            for _ in range(gat_layers):
                self.gat_layers.append(GraphAttentionLayer(hidden_dim, hidden_dim, n_heads, dropout))
            # Project back from nodes
            self.gat_node_unproj = nn.Linear(self.n_gat_nodes * hidden_dim, hidden_dim)

        self.pert_encoder = nn.Sequential(
            nn.Linear(pert_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )

        if use_cross_attn:
            self.cross_attention = CrossAttentionModule(
                hidden_dim, hidden_dim, hidden_dim, n_heads, dropout
            )
            fusion_input_dim = hidden_dim
        else:
            fusion_input_dim = hidden_dim * 2

        fusion_dim = fusion_input_dim
        self.fusion_dim = ((fusion_dim + n_heads - 1) // n_heads) * n_heads
        if self.fusion_dim != fusion_dim:
            self.fusion_proj = nn.Linear(fusion_dim, self.fusion_dim)
        else:
            self.fusion_proj = nn.Identity()

        # Multi-token transformer setup
        self.token_dim = ((self.fusion_dim // self.n_tokens + n_heads - 1) // n_heads) * n_heads
        self.token_proj = nn.Linear(self.fusion_dim, self.n_tokens * self.token_dim)
        self.pos_embedding = nn.Parameter(torch.randn(1, self.n_tokens, self.token_dim) * 0.02)

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=self.token_dim,
            nhead=n_heads,
            dim_feedforward=self.token_dim * 4,
            dropout=ffn_dropout,
            activation='gelu',
            batch_first=True,
            norm_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=n_layers)

        # Project back from tokens
        self.token_unproj = nn.Linear(self.n_tokens * self.token_dim, self.fusion_dim)

        self.fusion = nn.Sequential(
            nn.Linear(self.fusion_dim, hidden_dim),
            nn.LayerNorm(hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        )

        self.output_residuals = nn.ModuleList([
            ProteinResidualBlock(hidden_dim, dropout) for _ in range(1)
        ])

        self.output = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim//2),
            nn.LayerNorm(hidden_dim//2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim//2, input_dim)
        )

        self.perturbation_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim//2),
            nn.LayerNorm(hidden_dim//2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim//2, pert_dim)
        )

        self.apply(self._init_weights)
```

```
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)

    def forward(self, x, pert):
        batch_size = x.size(0)
        protein_feat = self.protein_encoder(x)

        for residual_block in self.protein_residuals:
            protein_feat = residual_block(protein_feat)

        if self.use_gat:
            # Project to multiple nodes: [batch, hidden_dim] -> [batch, n_nodes, hidden_dim]
            nodes = self.gat_node_proj(protein_feat)
            nodes = nodes.view(batch_size, self.n_gat_nodes, self.hidden_dim)
            for gat_layer in self.gat_layers:
                nodes = gat_layer(nodes)
                nodes = F.gelu(nodes)
            # Flatten and project back: [batch, n_nodes, hidden_dim] -> [batch, hidden_dim]
            nodes = nodes.view(batch_size, -1)
            protein_feat = self.gat_node_unproj(nodes)

        pert_feat = self.pert_encoder(pert)

        if self.use_cross_attn:
            fusion_input = self.cross_attention(protein_feat, pert_feat)
        else:
            fusion_input = torch.cat([protein_feat, pert_feat], dim=1)

        fusion_input = self.fusion_proj(fusion_input)

        # Project and reshape into tokens: [batch, fusion_dim] -> [batch, n_tokens, token_dim]
        tokens = self.token_proj(fusion_input)
        tokens = tokens.view(batch_size, self.n_tokens, self.token_dim)

        # Add positional embeddings
        tokens = tokens + self.pos_embedding

        # Apply transformer encoder
        x_trans = self.transformer(tokens)

        # Flatten tokens and project back: [batch, n_tokens, token_dim] -> [batch, fusion_dim]
        x_trans = x_trans.view(batch_size, -1)
        x_trans = self.token_unproj(x_trans)

        fused = self.fusion(x_trans)

        for residual_block in self.output_residuals:
            fused = residual_block(fused)

        output = self.output(fused)
        pert_pred = self.perturbation_head(fused)

        return output, pert_pred

def train_model(model, train_loader, optimizer, scheduler, device, aux_weight=0.1):
    model.train()
    total_loss = 0
    accumulation_steps = 4
    optimizer.zero_grad()

    for i, batch in enumerate(train_loader):
        baseline_expr, pert, target_expr = batch
        baseline_expr, pert, target_expr = baseline_expr.to(device), pert.to(device), target_expr.to(device)

        output, pert_pred = model(baseline_expr, pert)

        main_loss = F.mse_loss(output, target_expr)
        aux_loss = F.mse_loss(pert_pred, pert)
        loss = main_loss + aux_weight * aux_loss

        loss = loss / accumulation_steps
        loss.backward()

        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

        total_loss += loss.item() * accumulation_steps

    return total_loss / len(train_loader)

def evaluate_model(model, test_loader, device, aux_weight=0.1):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    all_perts = []
    all_pert_preds = []

    with torch.no_grad():
        for batch in test_loader:
            baseline_expr, pert, target_expr = batch
            baseline_expr, pert, target_expr = baseline_expr.to(device), pert.to(device), target_expr.to(device)

            output, pert_pred = model(baseline_expr, pert)

            # Compare predictions with target perturbed expression
```

```
            main_loss = F.mse_loss(output, target_expr)
            aux_loss = F.mse_loss(pert_pred, pert)
            loss = main_loss + aux_weight * aux_loss

            total_loss += loss.item()
            all_targets.append(target_expr.cpu().numpy())
            all_predictions.append(output.cpu().numpy())
            all_perts.append(pert.cpu().numpy())
            all_pert_preds.append(pert_pred.cpu().numpy())

    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)
    all_pert_preds = np.concatenate(all_pert_preds, axis=0)

    r2 = r2_score(all_targets, all_predictions)
    if np.isnan(r2):
        r2 = 0.0

    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pearson = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pearson):
            pearson = 0.0
    else:
        pearson = 0.0

    pert_r2 = r2_score(all_perts, all_pert_preds)
    if np.isnan(pert_r2):
        pert_r2 = 0.0

    return {
        'loss': total_loss / len(test_loader),
        'r2': r2,
        'pearson': pearson,
        'pert_r2': pert_r2
    }

def calculate_metrics(pred, true, control_baseline=None, perturbations=None, scaler=None):
    n_samples, n_proteins = true.shape
    mse = np.mean((pred - true) ** 2)
    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0

    true_mean_overall = np.mean(true)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_overall) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0

    if control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]

                if scaler is not None:
                    true_pert_inv = scaler.inverse_transform(true_pert)
                    control_baseline_inv = scaler.inverse_transform(control_baseline)
                    true_mean_pert = np.mean(true_pert_inv, axis=0)
                    control_mean = np.mean(control_baseline_inv, axis=0)
                else:
                    true_mean_pert = np.mean(true_pert, axis=0)
                    control_mean = np.mean(control_baseline, axis=0)

                lfc = np.log2((true_mean_pert + epsilon) / (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_proteins)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_proteins, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
                    pred_de_flat = pred_de.flatten()
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_overall = np.mean(true_de)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
```

```
                    ss_tot_de = np.sum((true_de - true_de_mean_overall) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            mse_de = np.mean((pred[de_mask] - true[de_mask]) ** 2)
            de_genes_indices = np.where(np.any(de_mask, axis=0))[0]
            if len(de_genes_indices) > 0:
                true_de = true[:, de_genes_indices]
                pred_de = pred[:, de_genes_indices]
                pred_de_flat = pred_de.flatten()
                true_de_flat = true_de.flatten()
                if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                    pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                    if np.isnan(pcc_de):
                        pcc_de = 0.0
                else:
                    pcc_de = 0.0
            else:
                pcc_de = 0.0
            true_de_mean = np.mean(true[de_mask])
            ss_res_de = np.sum((pred[de_mask] - true[de_mask]) ** 2)
            ss_tot_de = np.sum((true[de_mask] - true_de_mean) ** 2)
            if ss_tot_de > 1e-10:
                r2_de = 1.0 - (ss_res_de / ss_tot_de)
            else:
                r2_de = 0.0
            if np.isnan(r2_de):
                r2_de = 0.0
        else:
            mse_de = pcc_de = r2_de = np.nan

    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }
def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(list(set(train_dataset.perturbation_names + test_dataset.perturbation_names)))
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []

    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))

    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)

    for i, pair in enumerate(train_dataset.pairs):
        pert_idx = pair['perturbed_idx']
        train_dataset.pairs[i]['perturbation'] = train_dataset.perturbations[pert_idx]

    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(np.float32)

        for i, pair in enumerate(test_dataset.pairs):
            pert_idx = pair['perturbed_idx']
            test_dataset.pairs[i]['perturbation'] = test_dataset.perturbations[pert_idx]
        test_dataset._pert_encoding_standardized = True
```

```
    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True
    print_log(f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names

def objective(trial, timestamp):
    global train_dataset, test_dataset, device

    params = {
        'n_hidden': trial.suggest_int('n_hidden', 128, 512, step=8),
        'n_layers': trial.suggest_int('n_layers', 2, 4),
        'n_heads': trial.suggest_categorical('n_heads', [2, 4, 8]),
        'dropout': trial.suggest_float('dropout', 0.05, 0.3),
        'attention_dropout': trial.suggest_float('attention_dropout', 0.05, 0.2),
        'ffn_dropout': trial.suggest_float('ffn_dropout', 0.05, 0.2),
        'aux_weight': trial.suggest_float('aux_weight', 0.01, 0.2),
        'learning_rate': trial.suggest_float('learning_rate', 1e-5, 1e-3, log=True),
        'weight_decay': trial.suggest_float('weight_decay', 1e-6, 1e-4, log=True),
        'batch_size': trial.suggest_categorical('batch_size', [64, 128, 256, 512]),
        'use_gat': trial.suggest_categorical('use_gat', [True, False]),
        'gat_layers': trial.suggest_int('gat_layers', 1, 3),
        'use_cross_attn': trial.suggest_categorical('use_cross_attn', [True, False])
    }

    model = EnhancedProteinModel(
        input_dim=train_dataset.protein_data.shape[1],
        pert_dim=train_dataset.perturbations.shape[1],
        hidden_dim=params['n_hidden'],
        n_layers=params['n_layers'],
        n_heads=params['n_heads'],
        dropout=params['dropout'],
        attention_dropout=params['attention_dropout'],
        ffn_dropout=params['ffn_dropout'],
        use_gat=params['use_gat'],
        gat_layers=params['gat_layers'],
        use_cross_attn=params['use_cross_attn']
    ).to(device)

    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=params['learning_rate'],
        weight_decay=params['weight_decay']
    )

    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer,
        T_0=8,
        T_mult=2,
        eta_min=1e-6
    )

    train_loader = DataLoader(
        train_dataset,
        batch_size=params['batch_size'],
        shuffle=True,
        num_workers=4,
        pin_memory=True
    )

    test_loader = DataLoader(
        test_dataset,
        batch_size=min(params['batch_size'] * 2, 512),
        shuffle=False,
        num_workers=2,
        pin_memory=True
    )

    best_val_loss = float('inf')
    patience = 12
    patience_counter = 0
    max_epochs = 120

    for epoch in range(max_epochs):
        train_loss = train_model(model, train_loader, optimizer, scheduler, device,
                                 aux_weight=params['aux_weight'])

        val_metrics = evaluate_model(model, test_loader, device,
                                     aux_weight=params['aux_weight'])

        if val_metrics['loss'] < best_val_loss:
            best_val_loss = val_metrics['loss']
            patience_counter = 0
            torch.save(model.state_dict(), f'best_protein_model_trial_{trial.number}_{timestamp}.pt')
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break

        trial.report(val_metrics['loss'], epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

    return best_val_loss

def evaluate_and_save_model(model, test_loader, device, save_path, common_genes_info=None, scaler=None, best_params=None):
    model.eval()
    all_predictions = []
    all_targets = []
    all_baselines = []
    all_perts = []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc='Evaluating'):
            baseline_expr, pert, target_expr = batch
```

```
            baseline_expr, pert, target_expr = baseline_expr.to(device), pert.to(device), target_expr.to(device)

            output, _ = model(baseline_expr, pert)

            all_predictions.append(output.cpu().numpy())
            all_targets.append(target_expr.cpu().numpy())
            all_baselines.append(baseline_expr.cpu().numpy())
            all_perts.append(pert.cpu().numpy())

    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)
    all_baselines = np.concatenate(all_baselines, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)

    control_baseline = all_baselines
    results = calculate_metrics(all_predictions, all_targets, control_baseline=control_baseline, perturbations=all_perts, scaler=scaler)

    torch.save({
        'model_state_dict': model.state_dict(),
        'evaluation_results': results,
        'predictions': all_predictions,
        'targets': all_targets,
        'baselines': all_baselines,
        'protein_names': common_genes_info['genes'] if common_genes_info is not None else None,
        'common_genes_info': common_genes_info,
        'scaler': scaler,
        'best_params': best_params,
        'model_config': {
            'input_dim': model.input_dim,
            'pert_dim': model.pert_dim,
            'hidden_dim': getattr(model, 'hidden_dim', 256),
            'n_layers': getattr(model, 'n_layers', 3),
            'n_heads': getattr(model, 'n_heads', 4),
            'dropout': getattr(model, 'dropout', 0.1),
            'use_gat': getattr(model, 'use_gat', True),
            'use_cross_attn': getattr(model, 'use_cross_attn', True)
        },
        'perturbation_encoding': {
            'pert_dim': model.pert_dim,
            'all_pert_names': all_pert_names
        }
    }, save_path)

    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']]
    })

    print_log("\nProtein Model Evaluation Results:")
    print_log(metrics_df.to_string(index=False, float_format=lambda x: '{:.6f}'.format(x)))
    print_log(f"\nModel and evaluation results saved to: {save_path}")

    return results

def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, scaler

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print_log(f'Protein Model Training started at: {timestamp}')

    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print_log(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print_log(f'Warning: GPU {gpu_id} not available, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print_log(f'Using GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print_log(f'Using default GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print_log('CUDA not available, using CPU')

    print_log('Loading CITE-seq protein data...')
    train_path = "./datasets/PapalexiSatija2021_train.h5ad"
    test_path = "./datasets/PapalexiSatija2021_test.h5ad"

    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(f"Data files not found: {train_path} or {test_path}")

    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)

    print_log(f'Training data shape: {train_adata.shape}')
    print_log(f'Test data shape: {test_adata.shape}')

    print_log("Processing protein consistency...")
    train_proteins = set(train_adata.var_names)
    test_proteins = set(test_adata.var_names)
    common_proteins = list(train_proteins & test_proteins)

    print_log(f"Training proteins: {len(train_proteins)}")
    print_log(f"Test proteins: {len(test_proteins)}")
    print_log(f"Common proteins: {len(common_proteins)}")

    common_proteins.sort()
    train_protein_idx = [train_adata.var_names.get_loc(protein) for protein in common_proteins]
    test_protein_idx = [test_adata.var_names.get_loc(protein) for protein in common_proteins]

    if scipy.sparse.issparse(train_adata.X):
```

```
    train_data = train_adata.X[:, train_protein_idx].toarray()
else:
    train_data = train_adata.X[:, train_protein_idx]

train_data = np.maximum(train_data, 0)
train_data = np.maximum(train_data, 1e-10)
train_data = np.arcsinh(train_data / 5.0)

scaler = RobustScaler()
train_data = scaler.fit_transform(train_data)
train_data = np.clip(train_data, -5, 5)

common_genes_info = {
    'genes': common_proteins,
    'train_idx': train_protein_idx,
    'test_idx': test_protein_idx
}

train_dataset = CITEProteinDataset(
    train_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=True,
    is_train=True,
    common_genes_info=common_genes_info
)

test_dataset = CITEProteinDataset(
    test_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info
)

pert_dim, all_pert_names = standardize_perturbation_encoding(train_dataset, test_dataset)

optuna.logging.set_verbosity(optuna.logging.WARNING)
study = optuna.create_study(
    direction='minimize',
    sampler=optuna.samplers.TPESampler(seed=42),
    pruner=optuna.pruners.MedianPruner(
        n_startup_trials=5,
        n_warmup_steps=5,
        interval_steps=1
    )
)

print_log('Starting hyperparameter optimization...')
study.optimize(lambda trial: objective(trial, timestamp), n_trials=30)

best_params = study.best_params
print_log('Best parameters:')
for key, value in best_params.items():
    print_log(f'{key}: {value}')

final_model = EnhancedProteinModel(
    input_dim=train_dataset.protein_data.shape[1],
    pert_dim=pert_dim,
    hidden_dim=best_params['n_hidden'],
    n_layers=best_params['n_layers'],
    n_heads=best_params['n_heads'],
    dropout=best_params['dropout'],
    attention_dropout=best_params['attention_dropout'],
    ffn_dropout=best_params['ffn_dropout'],
    use_gat=best_params['use_gat'],
    gat_layers=best_params['gat_layers'],
    use_cross_attn=best_params['use_cross_attn']
).to(device)

train_loader = DataLoader(
    train_dataset,
    batch_size=best_params['batch_size'],
    shuffle=True,
    num_workers=4,
    pin_memory=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=min(best_params['batch_size'] * 2, 512),
    shuffle=False,
    num_workers=2,
    pin_memory=True
)

optimizer = torch.optim.AdamW(
    final_model.parameters(),
    lr=best_params['learning_rate'],
    weight_decay=best_params['weight_decay']
)

scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer,
    T_0=8,
    T_mult=2,
    eta_min=1e-6
)

print_log('Training final protein model...')
best_loss = float('inf')
best_model = None
max_epochs = 200
patience = 20
```

```
    patience_counter = 0

    for epoch in range(max_epochs):
        train_loss = train_model(final_model, train_loader, optimizer, scheduler, device,
                                 aux_weight=best_params['aux_weight'])
        eval_metrics = evaluate_model(final_model, test_loader, device,
                                      aux_weight=best_params['aux_weight'])

        if (epoch + 1) % 20 == 0:
            print_log(f'Epoch {epoch+1}/{max_epochs}:')
            print_log(f'Training Loss: {train_loss:.4f}')
            print_log(f'Test Loss: {eval_metrics["loss"]:.4f}')
            print_log(f'R2 Score: {eval_metrics["r2"]:.4f}')
            print_log(f'Pearson Correlation: {eval_metrics["pearson"]:.4f}')
            print_log(f'Perturbation R2: {eval_metrics["pert_r2"]:.4f}')

        if eval_metrics["loss"] < best_loss:
            best_loss = eval_metrics["loss"]
            patience_counter = 0
            best_model = final_model.state_dict()
            torch.save({
                'epoch': epoch,
                'model_state_dict': best_model,
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss': best_loss,
                'metrics': eval_metrics,
                'best_params': best_params,
                'scaler': scaler,
                'common_genes_info': common_genes_info,
                'perturbation_encoding': {
                    'pert_dim': pert_dim,
                    'all_pert_names': all_pert_names
                }
            }, f'cite_protein_best_model_{timestamp}.pt')
            print_log(f"Saved best protein model with loss: {best_loss:.4f}")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print_log(f'Early stopping at epoch {epoch+1}')
                break

    final_model.load_state_dict(best_model)
    print_log('Evaluating final protein model on test set...')
    results = evaluate_and_save_model(final_model, test_loader, device,
                                      f'cite_protein_final_model_{timestamp}.pt',
                                      common_genes_info, scaler, best_params, all_pert_names)

    best_params_str = str(best_params)
    if len(best_params_str) > 50:
        best_params_str = best_params_str[:50] + '...'

    results_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']],
        'Best_Params': [best_params_str] * 6
    })

    results_df.to_csv(f'cite_protein_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(index=False, float_format=lambda x: '{:.6f}'.format(x)))

    return results_df

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='CITE-seq Protein Model Training')
    parser.add_argument('--gpu', type=int, default=None,
                        help='Specify GPU ID (0-7)')
    parser.add_argument('--list-gpus', action='store_true',
                        help='List available GPUs and exit')
    args = parser.parse_args()

    if args.list_gpus:
        if torch.cuda.is_available():
            gpu_count = torch.cuda.device_count()
            print(f'Available GPUs: {gpu_count}')
            for i in range(gpu_count):
                print(f'GPU {i}: {torch.cuda.get_device_name(i)}')
        else:
            print('CUDA not available')
        exit(0)

    print("=" * 60)
    print("CITE-seq Protein Model Training")
    print("=" * 60)

    if args.gpu is not None:
        print(f"Using GPU: {args.gpu}")
    else:
        print("Using default GPU settings")

    results_df = main(gpu_id=args.gpu)
```

D.7 *ChromDDPM* for Liscovitch-Brauer scATAC-seq

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import scanpy as sc
```

```
import scipy
import anndata
from torch.utils.data import Dataset, DataLoader, random_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from tqdm import tqdm
import os
import pandas as pd
from sklearn.metrics import r2_score
from scipy.stats import pearsonr
import optuna
import optuna.logging
from IPython.display import display
import time
from datetime import datetime
import argparse
import math
import warnings
import random
import copy  # [OK] for deep copy of best model

warnings.filterwarnings('ignore')


def print_log(message):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {message}")


def set_global_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)


train_adata = None
test_adata = None
train_dataset = None
test_dataset = None
device = None
pca_model = None


class ATACDataset(Dataset):
    def __init__(self, adata, perturbation_key='perturbation', scaler=None, pca_model=None,
                 pca_dim=128, fit_pca=False, augment=False, is_train=True,
                 common_genes_info=None):
        self.adata = adata
        self.perturbation_key = perturbation_key
        self.augment = augment
        self.training = True
        self.pca_dim = pca_dim
        self.is_train = is_train
        self.common_genes_info = common_genes_info

        # gene subset
        if common_genes_info is not None:
            if is_train:
                gene_idx = common_genes_info['train_idx']
            else:
                gene_idx = common_genes_info['test_idx']
            if scipy.sparse.issparse(adata.X):
                data = adata.X[:, gene_idx].toarray()
            else:
                data = adata.X[:, gene_idx]
        else:
            if scipy.sparse.issparse(adata.X):
                data = adata.X.toarray()
            else:
                data = adata.X

        # pre-process expression
        data = np.maximum(data, 0)
        data = np.maximum(data, 1e-10)
        data_mean_before = np.mean(data)
        data_std_before = np.std(data)
        data_max_before = np.max(data)
        data_min_before = np.min(data)

        is_log1p_already = (data_max_before < 20.0 and data_min_before >= 0.0)
        is_standardized_already = (
            abs(data_mean_before) < 0.5 and 0.5 < data_std_before < 2.0)

        if not is_log1p_already:
            data = np.log1p(data)

        if scaler is None:
            if is_standardized_already:
                self.scaler = StandardScaler()
                self.scaler.mean_ = np.zeros(data.shape[1])
                self.scaler.scale_ = np.ones(data.shape[1])
                self.scaler.var_ = np.ones(data.shape[1])
                self.scaler.n_features_in_ = data.shape[1]
            else:
                self.scaler = StandardScaler()
                data = self.scaler.fit_transform(data)
        else:
            self.scaler = scaler
            if not is_standardized_already:
                data = self.scaler.transform(data)

        data = np.clip(data, -10, 10)
        data = data / 10.0
```

```
        # store data
        self.original_expression_data = data.copy()
        self.expression_data = data
        self.n_genes = data.shape[1]
        self.pca = pca_model if pca_model is not None else None
        self.pca_dim = pca_dim if pca_model is not None else None

        # perturbation encoding
        self.perturbation_names = [
            x for x in adata.obs[perturbation_key].unique() if pd.notnull(x)]
        self.perturbations = pd.get_dummies(adata.obs[perturbation_key]).values
        print(
            f"{'train' if is_train else 'test'} set perturbation dimension: {self.perturbations.shape[1]}")

        perturbation_labels = adata.obs[perturbation_key].astype(str).values
        perturbation_labels = np.array(perturbation_labels, dtype='U')
        lower_labels = np.char.lower(perturbation_labels)
        negctrl_mask = np.char.find(lower_labels, 'negctrl') >= 0
        control_mask = (lower_labels == 'control') | (
            lower_labels == 'ctrl') | negctrl_mask
        control_indices = np.where(control_mask)[0]
        non_control_indices = np.where(~control_mask)[0]

        self._pert_to_indices = {}
        for pert_name in self.perturbation_names:
            pert_mask = perturbation_labels == pert_name
            pert_indices = np.where(pert_mask)[0]
            pert_indices = pert_indices[~np.isin(
                pert_indices, control_indices)]
            if len(pert_indices) > 0:
                self._pert_to_indices[pert_name] = pert_indices

        self._non_control_pert_names = [
            name for name in self.perturbation_names
            if isinstance(name, str) and name not in ['control', 'Control', 'ctrl', 'Ctrl']
            and 'negctrl' not in name.lower()
        ]
        self._control_indices = control_indices
        self._non_control_indices = non_control_indices
        # [OK] for fast membership checks
        self._control_set = set(self._control_indices.tolist())

        if len(self._control_indices) == 0:
            if self.is_train:
                raise ValueError(
                    "CRITICAL ERROR: Training set has NO control samples! "
                    "Control samples are REQUIRED for perturbation prediction. "
                    "This model performs control -> perturbed prediction, NOT autoencoder. "
                    "Please ensure your training data includes control samples (labeled as 'control' or containing 'NegCtrl')."
                )
            else:
                raise ValueError(
                    "CRITICAL ERROR: Test set has NO control samples! "
                    "For unseen perturbation prediction, test set should use training set control baseline. "
                    "Please ensure training set has control samples."
                )

        self.rng = np.random.RandomState(42)
        if not self.is_train:
            self._create_fixed_pairs_for_test()

    def _create_fixed_pairs_for_test(self):
        n = len(self.adata)
        self.pairs = [None] * n
        for idx in range(n):
            if idx in self._control_set:
                baseline_idx = idx
                if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                    pert_name = self._non_control_pert_names[idx % len(
                        self._non_control_pert_names)]
                    if pert_name in self._pert_to_indices and len(self._pert_to_indices[pert_name]) > 0:
                        pert_indices = self._pert_to_indices[pert_name]
                        target_idx = int(
                            pert_indices[(idx // len(self._non_control_pert_names)) % len(pert_indices)])
                    else:
                        target_idx = int(
                            self._non_control_indices[idx % len(self._non_control_indices)])
                else:
                    alternative_controls = self._control_indices[self._control_indices != idx]
                    if len(alternative_controls) > 0:
                        target_idx = int(
                            alternative_controls[idx % len(alternative_controls)])
                    else:
                        target_idx = idx
            else:
                if len(self._control_indices) > 0:
                    baseline_idx = int(
                        self._control_indices[idx % len(self._control_indices)])
                else:
                    baseline_idx = idx
                target_idx = idx
            self.pairs[idx] = (baseline_idx, target_idx)

    def __len__(self):
        return len(self.adata)

    def __getitem__(self, idx):
        x_current = self.expression_data[idx]
        pert = self.perturbations[idx]

        if self.is_train:
            if idx in self._control_set:
                x_baseline = x_current
                if len(self._non_control_pert_names) > 0 and len(self._non_control_indices) > 0:
                    pert_name = self.rng.choice(self._non_control_pert_names)
```

```
                    if pert_name in self._pert_to_indices and len(self._pert_to_indices[pert_name]) > 0:
                        pert_indices = self._pert_to_indices[pert_name]
                        target_idx = int(self.rng.choice(pert_indices))
                        x_target = self.original_expression_data[target_idx]
                        pert_target = self.perturbations[target_idx]
                    else:
                        target_idx = int(self.rng.choice(
                            self._non_control_indices))
                        x_target = self.original_expression_data[target_idx]
                        pert_target = self.perturbations[target_idx]
                else:
                    alternative_controls = self._control_indices[self._control_indices != idx]
                    if len(alternative_controls) > 0:
                        target_idx = int(self.rng.choice(alternative_controls))
                        x_target = self.original_expression_data[target_idx]
                        pert_target = self.perturbations[target_idx]
                    else:
                        x_target = self.original_expression_data[idx]
                        pert_target = pert
            else:
                if len(self._control_indices) > 0:
                    baseline_idx = int(self.rng.choice(self._control_indices))
                    x_baseline = self.expression_data[baseline_idx]
                else:
                    x_baseline = x_current
                x_target = self.original_expression_data[idx]
                pert_target = pert
        else:
            baseline_idx, target_idx = self.pairs[idx]
            if baseline_idx == idx and idx in self._control_set:
                x_baseline = x_current
            else:
                x_baseline = self.expression_data[baseline_idx]
            x_target = self.original_expression_data[target_idx]
            if target_idx == idx:
                pert_target = pert
            else:
                pert_target = self.perturbations[target_idx]

        x_target_delta = x_target - x_baseline

        if self.augment and self.training:
            noise = np.random.normal(0, 0.05, x_baseline.shape)
            x_baseline = x_baseline + noise
            mask = np.random.random(x_baseline.shape) > 0.05
            x_baseline = x_baseline * mask

        return torch.FloatTensor(x_baseline), torch.FloatTensor(pert_target), torch.FloatTensor(x_target_delta)


class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(
            half_dim, device=device) * -embeddings)
        time = time.float()
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        if embeddings.shape[-1] != self.dim:
            if embeddings.shape[-1] < self.dim:
                padding = torch.zeros(
                    embeddings.shape[0], self.dim - embeddings.shape[-1], device=device)
                embeddings = torch.cat([embeddings, padding], dim=-1)
            else:
                embeddings = embeddings[:, :self.dim]
        return embeddings


class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, time_emb_dim, dropout=0.1):
        super().__init__()
        self.time_mlp = nn.Linear(time_emb_dim, out_channels)
        self.block1 = nn.Sequential(
            nn.Linear(in_channels, out_channels),
            nn.LayerNorm(out_channels),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.block2 = nn.Sequential(
            nn.Linear(out_channels, out_channels),
            nn.LayerNorm(out_channels),
            nn.GELU(),
            nn.Dropout(dropout)
        )
        self.res_conv = nn.Linear(
            in_channels, out_channels) if in_channels != out_channels else nn.Identity()

    def forward(self, x, time_emb):
        h = self.block1(x)
        time_emb = self.time_mlp(time_emb)
        h = h + time_emb
        h = self.block2(h)
        return h + self.res_conv(x)


class ConditionalDiffusionModel(nn.Module):
    def __init__(self, input_dim, output_dim, train_pert_dim, test_pert_dim, hidden_dim=512,
                 n_layers=4, n_heads=8, dropout=0.1, time_emb_dim=128,
                 diffusion_steps=1000, pert_emb_dim=64,
```

```
        model_in_dim=128, bottleneck_dims=(2048, 512), use_bottleneck=True):
    super(ConditionalDiffusionModel, self).__init__()
    self.input_dim = input_dim
    self.output_dim = output_dim
    self.model_in_dim = model_in_dim
    self.train_pert_dim = train_pert_dim
    self.test_pert_dim = test_pert_dim
    assert train_pert_dim == test_pert_dim, "After standardization, train_pert_dim and test_pert_dim must be equal"

    # store hyperparams for saving/loading
    self.n_layers = n_layers
    self.n_heads = n_heads
    self.dropout = dropout
    self.diffusion_steps = diffusion_steps
    self.time_emb_dim = time_emb_dim
    self.pert_emb_dim = pert_emb_dim
    self.use_bottleneck = use_bottleneck
    self.bottleneck_dims = bottleneck_dims
    self.model_in_dim = model_in_dim

    self.hidden_dim = ((hidden_dim + n_heads - 1) // n_heads) * n_heads

    self.time_embeddings = SinusoidalPositionEmbeddings(time_emb_dim)
    self.time_mlp = nn.Sequential(
        nn.Linear(time_emb_dim, time_emb_dim),
        nn.GELU()
    )

    if self.use_bottleneck:
        self.input_proj = nn.Sequential(
            nn.Linear(self.input_dim, bottleneck_dims[0]),
            nn.LayerNorm(bottleneck_dims[0]),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(bottleneck_dims[0], bottleneck_dims[1]),
            nn.LayerNorm(bottleneck_dims[1]),
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(bottleneck_dims[1], self.model_in_dim),
            nn.LayerNorm(self.model_in_dim),
        )
    else:
        self.input_proj = nn.Sequential(
            nn.Linear(self.input_dim, self.model_in_dim),
            nn.LayerNorm(self.model_in_dim),
            nn.GELU()
        )

    self.model_input_proj = nn.Linear(self.model_in_dim, self.hidden_dim)

    self.pert_encoder = nn.Sequential(
        nn.Linear(train_pert_dim, pert_emb_dim),
        nn.LayerNorm(pert_emb_dim),
        nn.GELU(),
        nn.Dropout(dropout)
    )

    self.cond_proj = nn.Linear(
        pert_emb_dim + time_emb_dim, self.hidden_dim)

    mlp_layers = []
    for _ in range(n_layers):
        mlp_layers.extend([
            nn.Linear(self.hidden_dim, self.hidden_dim),
            nn.LayerNorm(self.hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout)
        ])
    self.mlp = nn.Sequential(*mlp_layers) if mlp_layers else nn.Identity()

    self.output_proj = nn.Sequential(
        nn.Linear(self.hidden_dim, self.hidden_dim),
        nn.LayerNorm(self.hidden_dim),
        nn.GELU(),
        nn.Dropout(dropout),
        nn.Linear(self.hidden_dim, output_dim)
    )

    self.noise_proj = nn.Sequential(
        nn.Linear(self.hidden_dim, self.hidden_dim),
        nn.LayerNorm(self.hidden_dim),
        nn.GELU(),
        nn.Dropout(dropout),
        nn.Linear(self.hidden_dim, self.model_in_dim)
    )

    self.output_head = nn.Sequential(
        nn.Linear(self.model_in_dim, self.hidden_dim),
        nn.LayerNorm(self.hidden_dim),
        nn.GELU(),
        nn.Dropout(dropout),
        nn.Linear(self.hidden_dim, output_dim)
    )

    self.pert_head_shared = nn.Sequential(
        nn.Linear(self.hidden_dim, self.hidden_dim // 2),
        nn.LayerNorm(self.hidden_dim // 2),
        nn.GELU(),
        nn.Dropout(dropout)
    )
    self.perturbation_head = nn.Linear(
        self.hidden_dim // 2, train_pert_dim)

    self.apply(self._init_weights)
```

```
    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                nn.init.zeros_(module.bias)
        elif isinstance(module, nn.LayerNorm):
            nn.init.ones_(module.weight)
            nn.init.zeros_(module.bias)

    def forward(self, x, pert, timestep, is_train=True):
        assert x.dim() == 2 and x.size(1) == self.input_dim, \
            f"Expected x shape [B, {self.input_dim}], got {x.shape}"

        x_proj = self.input_proj(x)
        time_emb = self.time_embeddings(timestep)
        time_emb = self.time_mlp(time_emb)
        pert_emb = self.pert_encoder(pert)
        cond_emb = torch.cat([pert_emb, time_emb], dim=-1)
        cond_emb = self.cond_proj(cond_emb)

        x_proj_model = self.model_input_proj(x_proj)
        x_cond = x_proj_model + cond_emb
        x_trans = self.mlp(x_cond)

        noise_pred_proj = self.noise_proj(x_trans)
        noise_pred = self.output_head(noise_pred_proj)
        # kept as-is; both are same tensor
        output = self.output_head(noise_pred_proj)

        x_pert_feat = self.pert_head_shared(x_trans)
        pert_pred = self.perturbation_head(x_pert_feat)

        return noise_pred, pert_pred, output


class DiffusionModel(nn.Module):
    def __init__(self, model, diffusion_steps=1000, beta_start=1e-4, beta_end=0.02):
        super().__init__()
        self.model = model
        self.diffusion_steps = diffusion_steps
        self.register_buffer('betas', torch.linspace(
            beta_start, beta_end, diffusion_steps))
        self.register_buffer('alphas', 1.0 - self.betas)
        self.register_buffer('alpha_bars', torch.cumprod(self.alphas, dim=0))

    def q_sample(self, x_start, t, noise=None):
        if noise is None:
            noise = torch.randn_like(x_start)
        sqrt_alpha_bar_t = torch.sqrt(self.alpha_bars[t]).view(-1, 1)
        sqrt_one_minus_alpha_bar_t = torch.sqrt(
            1 - self.alpha_bars[t]).view(-1, 1)
        return sqrt_alpha_bar_t * x_start + sqrt_one_minus_alpha_bar_t * noise

    def p_sample(self, x, pert, t, is_train=True):
        with torch.no_grad():
            pred_noise, _, _ = self.model(x, pert, t, is_train)
            alpha_t = self.alphas[t].view(-1, 1)
            alpha_bar_t = self.alpha_bars[t].view(-1, 1)
            beta_t = self.betas[t].view(-1, 1)

            eps = 1e-8
            sqrt_alpha_bar_t = torch.sqrt(torch.clamp(alpha_bar_t, min=eps))
            sqrt_one_minus_alpha_bar_t = torch.sqrt(
                torch.clamp(1 - alpha_bar_t, min=eps))
            sqrt_alpha_t = torch.sqrt(torch.clamp(alpha_t, min=eps))

            pred_x_start = (x - sqrt_one_minus_alpha_bar_t *
                            pred_noise) / sqrt_alpha_bar_t

            if torch.any(torch.isnan(pred_x_start)) or torch.any(torch.isinf(pred_x_start)):
                pred_x_start = torch.clamp(pred_x_start, min=-10.0, max=10.0)
                pred_x_start = torch.where(torch.isnan(pred_x_start) | torch.isinf(pred_x_start),
                                           x, pred_x_start)

            if t[0].item() > 0:
                noise = torch.randn_like(x)
                mean = (pred_x_start * sqrt_alpha_bar_t + x *
                        sqrt_one_minus_alpha_bar_t) / sqrt_alpha_t

                if torch.any(torch.isnan(mean)) or torch.any(torch.isinf(mean)):
                    mean = torch.clamp(mean, min=-10.0, max=10.0)
                    mean = torch.where(torch.isnan(
                        mean) | torch.isinf(mean), x, mean)

                result = mean + \
                    torch.sqrt(torch.clamp(beta_t, min=eps)) * noise

                if torch.any(torch.isnan(result)) or torch.any(torch.isinf(result)):
                    result = torch.clamp(result, min=-10.0, max=10.0)
                    result = torch.where(torch.isnan(
                        result) | torch.isinf(result), x, result)

                return result
            else:
                return pred_x_start

    def p_sample_loop(self, shape, pert, is_train=True):
        device = next(self.parameters()).device
        x = torch.randn(shape, device=device)
        for t in reversed(range(self.diffusion_steps)):
            t_tensor = torch.full(
                (shape[0],), t, device=device, dtype=torch.long)
            x = self.p_sample(x, pert, t_tensor, is_train)
        return x
```

```
    def p_sample_loop_from_baseline(self, x_baseline, pert, is_train=True, noise_scale=0.1):
        device = next(self.parameters()).device
        x = x_baseline + noise_scale * torch.randn_like(x_baseline)

        start_t = max(1, self.diffusion_steps // 4)
        for t in reversed(range(start_t)):
            t_tensor = torch.full(
                (x.shape[0],), t, device=device, dtype=torch.long)
            x = self.p_sample(x, pert, t_tensor, is_train)

            if torch.any(torch.isnan(x)) or torch.any(torch.isinf(x)):
                x = torch.clamp(x, min=-10.0, max=10.0)
                x = torch.where(torch.isnan(x) | torch.isinf(x), x_baseline, x)

        return x

    def forward(self, x_start, pert, is_train=True):
        batch_size = x_start.shape[0]
        device = x_start.device
        t = torch.randint(0, self.diffusion_steps,
                          (batch_size,), device=device)
        noise = torch.randn_like(x_start)
        x_noisy = self.q_sample(x_start, t, noise)
        pred_noise, pert_pred, _ = self.model(x_noisy, pert, t, is_train)
        return pred_noise, pert_pred, noise, t


def train_model(model, train_loader, optimizer, scheduler, device, aux_weight=0.1):
    model.train()
    total_loss = 0
    accumulation_steps = 4
    optimizer.zero_grad()
    for i, batch in enumerate(train_loader):
        x_baseline, pert, x_target_delta = batch
        x_baseline, pert, x_target_delta = x_baseline.to(
            device), pert.to(device), x_target_delta.to(device)
        x_target_abs = x_baseline + x_target_delta

        pred_noise, pert_pred, noise, t = model(
            x_target_abs, pert, is_train=True)
        diffusion_loss = F.mse_loss(pred_noise, noise)
        aux_loss = F.mse_loss(pert_pred, pert)
        loss = diffusion_loss + aux_weight * aux_loss

        loss = loss / accumulation_steps
        loss.backward()

        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

        total_loss += loss.item() * accumulation_steps

    return total_loss / len(train_loader)


def evaluate_model(model, test_loader, device, aux_weight=0.1):
    model.eval()
    total_loss = 0
    all_targets = []
    all_predictions = []
    all_perts = []
    all_pert_preds = []
    valid_batches = 0

    with torch.no_grad():
        for batch in test_loader:
            x_baseline, pert, x_target_delta = batch
            x_baseline, pert, x_target_delta = x_baseline.to(
                device), pert.to(device), x_target_delta.to(device)

            x_pred_abs = model.p_sample_loop_from_baseline(
                x_baseline, pert, is_train=False)

            if torch.any(torch.isnan(x_pred_abs)) or torch.any(torch.isinf(x_pred_abs)):
                x_pred_abs = x_baseline.clone()

            x_target_abs = x_baseline + x_target_delta
            diffusion_loss = F.mse_loss(x_pred_abs, x_target_abs)

            if torch.isnan(diffusion_loss) or torch.isinf(diffusion_loss):
                continue

            t_dummy = torch.randint(
                0, model.diffusion_steps, (x_baseline.shape[0],), device=device)
            _, pert_pred, _ = model.model(
                x_pred_abs, pert, t_dummy, is_train=False)

            if pert_pred.shape[1] == pert.shape[1]:
                aux_loss = F.mse_loss(pert_pred, pert)
                if torch.isnan(aux_loss) or torch.isinf(aux_loss):
                    aux_loss = torch.tensor(0.0, device=device)
            else:
                aux_loss = torch.tensor(0.0, device=device)

            loss = diffusion_loss + aux_weight * aux_loss

            if torch.isnan(loss) or torch.isinf(loss):
                continue

            total_loss += loss.item()
            valid_batches += 1
```

```
            all_targets.append(x_target_abs.cpu().numpy())
            all_predictions.append(x_pred_abs.cpu().numpy())
            if pert_pred.shape[1] == pert.shape[1]:
                all_perts.append(pert.cpu().numpy())
                all_pert_preds.append(pert_pred.cpu().numpy())

    if valid_batches == 0:
        return {
            'loss': float('inf'),
            'r2': 0.0,
            'pearson': 0.0,
            'pert_r2': 0.0
        }

    all_targets = np.concatenate(all_targets, axis=0)
    all_predictions = np.concatenate(all_predictions, axis=0)

    all_predictions = np.nan_to_num(
        all_predictions, nan=0.0, posinf=10.0, neginf=-10.0)
    all_targets = np.nan_to_num(
        all_targets, nan=0.0, posinf=10.0, neginf=-10.0)

    r2 = r2_score(all_targets, all_predictions)
    if np.isnan(r2) or np.isinf(r2):
        r2 = 0.0

    pred_flat = all_predictions.flatten()
    true_flat = all_targets.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pearson = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pearson) or np.isinf(pearson):
            pearson = 0.0
    else:
        pearson = 0.0

    if len(all_perts) > 0:
        all_perts = np.concatenate(all_perts, axis=0)
        all_pert_preds = np.concatenate(all_pert_preds, axis=0)
        all_pert_preds = np.nan_to_num(
            all_pert_preds, nan=0.0, posinf=1.0, neginf=-1.0)

        pert_r2 = r2_score(all_perts, all_pert_preds)
        if np.isnan(pert_r2) or np.isinf(pert_r2):
            pert_r2 = 0.0
    else:
        pert_r2 = 0.0

    return {
        'loss': total_loss / valid_batches if valid_batches > 0 else float('inf'),
        'r2': r2,
        'pearson': pearson,
        'pert_r2': pert_r2
    }


def calculate_metrics(pred, true, control_baseline=None, perturbations=None):
    n_samples, n_genes = true.shape
    mse = np.mean((pred - true) ** 2)

    pred_flat = pred.flatten()
    true_flat = true.flatten()
    if len(pred_flat) > 1 and np.std(pred_flat) > 1e-10 and np.std(true_flat) > 1e-10:
        pcc = pearsonr(true_flat, pred_flat)[0]
        if np.isnan(pcc):
            pcc = 0.0
    else:
        pcc = 0.0

    true_mean_overall = np.mean(true)
    ss_res = np.sum((true - pred) ** 2)
    ss_tot = np.sum((true - true_mean_overall) ** 2)
    if ss_tot > 1e-10:
        r2 = 1.0 - (ss_res / ss_tot)
    else:
        r2 = 0.0
    if np.isnan(r2):
        r2 = 0.0

    if control_baseline is not None and control_baseline.shape[0] > 0:
        epsilon = 1e-8
        if perturbations is not None and len(perturbations) == n_samples:
            pert_indices = np.argmax(perturbations, axis=1)
            unique_perts = np.unique(pert_indices)
            mse_de_list = []
            pcc_de_list = []
            r2_de_list = []
            for pert_idx in unique_perts:
                pert_mask = pert_indices == pert_idx
                if np.sum(pert_mask) < 2:
                    continue
                true_pert = true[pert_mask]
                pred_pert = pred[pert_mask]
                true_mean_pert = np.mean(true_pert, axis=0)
                control_mean = np.mean(control_baseline, axis=0)
                lfc = np.log2((true_mean_pert + epsilon) /
                              (control_mean + epsilon))
                lfc_abs = np.abs(lfc)
                K = min(20, n_genes)
                top_k_indices = np.argsort(lfc_abs)[-K:]
                de_mask = np.zeros(n_genes, dtype=bool)
                de_mask[top_k_indices] = True
                if np.any(de_mask):
                    true_de = true_pert[:, de_mask]
                    pred_de = pred_pert[:, de_mask]
                    mse_de_pert = np.mean((pred_de - true_de) ** 2)
```

```
                    pred_de_flat = pred_de.flatten()
                    true_de_flat = true_de.flatten()
                    if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                        pcc_de_pert = pearsonr(true_de_flat, pred_de_flat)[0]
                        if np.isnan(pcc_de_pert):
                            pcc_de_pert = 0.0
                    else:
                        pcc_de_pert = 0.0
                    true_de_mean_overall = np.mean(true_de)
                    ss_res_de = np.sum((true_de - pred_de) ** 2)
                    ss_tot_de = np.sum((true_de - true_de_mean_overall) ** 2)
                    if ss_tot_de > 1e-10:
                        r2_de_pert = 1.0 - (ss_res_de / ss_tot_de)
                    else:
                        r2_de_pert = 0.0
                    if np.isnan(r2_de_pert):
                        r2_de_pert = 0.0
                    mse_de_list.append(mse_de_pert)
                    pcc_de_list.append(pcc_de_pert)
                    r2_de_list.append(r2_de_pert)
            if len(mse_de_list) > 0:
                mse_de = np.mean(mse_de_list)
                pcc_de = np.mean(pcc_de_list)
                r2_de = np.mean(r2_de_list)
            else:
                mse_de = pcc_de = r2_de = np.nan
        else:
            mse_de = pcc_de = r2_de = np.nan
    else:
        std = np.std(true, axis=0)
        de_mask = np.abs(true - np.mean(true, axis=0)) > std
        if np.any(de_mask):
            mse_de = np.mean((pred[de_mask] - true[de_mask]) ** 2)
            pred_de_flat = pred[de_mask].flatten()
            true_de_flat = true[de_mask].flatten()
            if len(pred_de_flat) > 1 and np.std(pred_de_flat) > 1e-10 and np.std(true_de_flat) > 1e-10:
                pcc_de = pearsonr(true_de_flat, pred_de_flat)[0]
                if np.isnan(pcc_de):
                    pcc_de = 0.0
            else:
                pcc_de = 0.0
            true_de_mean = np.mean(true[de_mask])
            ss_res_de = np.sum((pred[de_mask] - true[de_mask]) ** 2)
            ss_tot_de = np.sum((true[de_mask] - true_de_mean) ** 2)
            if ss_tot_de > 1e-10:
                r2_de = 1.0 - (ss_res_de / ss_tot_de)
            else:
                r2_de = 0.0
            if np.isnan(r2_de):
                r2_de = 0.0
        else:
            mse_de = pcc_de = r2_de = np.nan

    return {
        'MSE': mse,
        'PCC': pcc,
        'R2': r2,
        'MSE_DE': mse_de,
        'PCC_DE': pcc_de,
        'R2_DE': r2_de
    }


def evaluate_and_save_model(model, test_loader, device, save_path='atac_diffusion_best.pt',
                            common_genes_info=None, pca_model=None, scaler=None, best_params=None):
    model.eval()
    all_predictions = []
    all_targets = []
    all_baselines = []
    all_perts = []

    with torch.no_grad():
        for batch in tqdm(test_loader, desc='Evaluating'):
            x_baseline, pert, x_target_delta = batch
            x_baseline, pert, x_target_delta = x_baseline.to(
                device), pert.to(device), x_target_delta.to(device)
            x_pred_abs = model.p_sample_loop_from_baseline(
                x_baseline, pert, is_train=False)
            x_target_abs = x_baseline + x_target_delta

            all_predictions.append(x_pred_abs.cpu().numpy())
            all_targets.append(x_target_abs.cpu().numpy())
            all_baselines.append(x_baseline.cpu().numpy())
            all_perts.append(pert.cpu().numpy())

    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)
    all_baselines = np.concatenate(all_baselines, axis=0)
    all_perts = np.concatenate(all_perts, axis=0)

    control_baseline = all_baselines
    results = calculate_metrics(all_predictions, all_targets,
                                control_baseline=control_baseline,
                                perturbations=all_perts)

    torch.save({
        'model_state_dict': model.state_dict(),
        'evaluation_results': results,
        'predictions': all_predictions,
        'targets': all_targets,
        'baselines': all_baselines,
        'gene_names': common_genes_info['genes'] if common_genes_info is not None else None,
        'pca_model': pca_model,
        'scaler': scaler,
        'best_params': best_params,
```

```
        'model_config': {
            'input_dim': model.model.input_dim,
            'output_dim': model.model.output_dim,
            'train_pert_dim': model.model.train_pert_dim,
            'test_pert_dim': model.model.test_pert_dim,
            'hidden_dim': model.model.hidden_dim,
            'n_layers': getattr(model.model, 'n_layers', 4),
            'n_heads': getattr(model.model, 'n_heads', 8),
            'dropout': getattr(model.model, 'dropout', 0.1),
            'diffusion_steps': getattr(model, 'diffusion_steps', 1000),
            'time_emb_dim': model.model.time_emb_dim,
            'pert_emb_dim': model.model.pert_emb_dim,
            'model_in_dim': getattr(model.model, 'model_in_dim', 128),
            'use_bottleneck': getattr(model.model, 'use_bottleneck', True),
            'bottleneck_dims': getattr(model.model, 'bottleneck_dims', (2048, 512))
        }
    }, save_path)

    metrics_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']]
    })
    print("\nEvaluation Results:")
    print(metrics_df.to_string(index=False,
          float_format=lambda x: '{:.6f}'.format(x)))
    print(f"\nModel and evaluation results saved to: {save_path}")
    return results


def standardize_perturbation_encoding(train_dataset, test_dataset=None, test_perturbation_names=None):
    if test_dataset is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and hasattr(test_dataset, '_pert_encoding_standardized'):
            if train_dataset._pert_encoding_standardized and test_dataset._pert_encoding_standardized:
                if train_dataset.perturbations.shape[1] == test_dataset.perturbations.shape[1]:
                    return train_dataset.perturbations.shape[1], sorted(
                        list(set(train_dataset.perturbation_names +
                                 test_dataset.perturbation_names))
                    )
        test_pert_names = test_dataset.perturbation_names
    elif test_perturbation_names is not None:
        if hasattr(train_dataset, '_pert_encoding_standardized') and train_dataset._pert_encoding_standardized:
            all_pert_names = set(
                train_dataset.perturbation_names + test_perturbation_names)
            actual_pert_dim = len(all_pert_names)
            if train_dataset.perturbations.shape[1] == actual_pert_dim:
                return actual_pert_dim, sorted(list(all_pert_names))
        test_pert_names = test_perturbation_names
    else:
        test_pert_names = []

    train_pert_dim = train_dataset.perturbations.shape[1]
    all_pert_names = set(train_dataset.perturbation_names + test_pert_names)
    all_pert_names = sorted(list(all_pert_names))

    train_pert_df = pd.DataFrame(train_dataset.adata.obs['perturbation'])
    train_pert_encoded = pd.get_dummies(train_pert_df['perturbation'])
    for pert_name in all_pert_names:
        if pert_name not in train_pert_encoded.columns:
            train_pert_encoded[pert_name] = 0
    train_pert_encoded = train_pert_encoded.reindex(
        columns=all_pert_names, fill_value=0)
    train_dataset.perturbations = train_pert_encoded.values.astype(np.float32)

    if test_dataset is not None:
        test_pert_df = pd.DataFrame(test_dataset.adata.obs['perturbation'])
        test_pert_encoded = pd.get_dummies(test_pert_df['perturbation'])
        for pert_name in all_pert_names:
            if pert_name not in test_pert_encoded.columns:
                test_pert_encoded[pert_name] = 0
        test_pert_encoded = test_pert_encoded.reindex(
            columns=all_pert_names, fill_value=0)
        test_dataset.perturbations = test_pert_encoded.values.astype(
            np.float32)
        test_dataset._pert_encoding_standardized = True

    actual_pert_dim = len(all_pert_names)
    train_dataset._pert_encoding_standardized = True

    print_log(
        f"Standardized perturbation encoding: {actual_pert_dim} dimensions")
    print_log(f"All perturbation types: {all_pert_names}")
    return actual_pert_dim, all_pert_names


def objective(trial, timestamp):
    global train_dataset, test_dataset, device, pca_model, test_adata

    params = {
        'pca_dim': 128,
        'n_hidden': trial.suggest_int('n_hidden', 256, 1024),
        'n_layers': trial.suggest_int('n_layers', 2, 6),
        'n_heads': trial.suggest_int('n_heads', 4, 8),
        'dropout': trial.suggest_float('dropout', 0.1, 0.3),
        'aux_weight': trial.suggest_float('aux_weight', 0.05, 0.15),
        'learning_rate': trial.suggest_float('learning_rate', 1e-4, 1e-3, log=True),
        'weight_decay': trial.suggest_float('weight_decay', 1e-5, 1e-4, log=True),
        'batch_size': trial.suggest_categorical('batch_size', [32, 64, 128]),
        'time_emb_dim': trial.suggest_int('time_emb_dim', 64, 256),
        'pert_emb_dim': trial.suggest_int('pert_emb_dim', 32, 128),
        'diffusion_steps': trial.suggest_categorical('diffusion_steps', [500, 1000, 2000])
    }

    if test_adata is not None:
        test_perturbation_names = [
```

```
        x for x in test_adata.obs['perturbation'].unique() if pd.notnull(x)]
else:
    if test_dataset is not None and hasattr(test_dataset, 'perturbation_names'):
        test_perturbation_names = test_dataset.perturbation_names
    else:
        test_perturbation_names = train_dataset.perturbation_names if hasattr(
            train_dataset, 'perturbation_names') else []

pert_dim, _ = standardize_perturbation_encoding(train_dataset, test_dataset=None,
                                                test_perturbation_names=test_perturbation_names)
n_genes = train_dataset.n_genes
print_log(
    f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")

base_model = ConditionalDiffusionModel(
    input_dim=n_genes,
    output_dim=n_genes,
    train_pert_dim=pert_dim,
    test_pert_dim=pert_dim,
    hidden_dim=params['n_hidden'],
    n_layers=params['n_layers'],
    n_heads=params['n_heads'],
    dropout=params['dropout'],
    time_emb_dim=params['time_emb_dim'],
    pert_emb_dim=params['pert_emb_dim'],
    diffusion_steps=params['diffusion_steps']
)
model = DiffusionModel(
    base_model, diffusion_steps=params['diffusion_steps']).to(device)

optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=params['learning_rate'],
    weight_decay=params['weight_decay']
)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer,
    T_0=10,
    T_mult=2,
    eta_min=1e-6
)

train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(
    train_dataset, [train_size, val_size])

train_loader = DataLoader(
    train_subset,
    batch_size=params['batch_size'],
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
val_loader = DataLoader(
    val_subset,
    batch_size=params['batch_size'],
    shuffle=False,
    num_workers=4,
    pin_memory=True
)

best_val_loss = float('inf')
patience = 15
patience_counter = 0
max_epochs = 150

for epoch in range(max_epochs):
    model.train()
    train_loss = 0
    for batch in tqdm(train_loader, desc=f'Epoch {epoch + 1}/{max_epochs}'):
        x_baseline, pert, x_target_delta = batch
        x_baseline, pert, x_target_delta = x_baseline.to(
            device), pert.to(device), x_target_delta.to(device)
        x_target_abs = x_baseline + x_target_delta

        optimizer.zero_grad()
        pred_noise, pert_pred, noise, t = model(
            x_target_abs, pert, is_train=True)
        diffusion_loss = F.mse_loss(pred_noise, noise)
        pert_loss = F.mse_loss(pert_pred, pert)
        loss = diffusion_loss + params['aux_weight'] * pert_loss
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(train_loader)

    model.eval()
    val_loss = 0
    val_batches_processed = 0
    max_val_batches = 50
    with torch.no_grad():
        for batch in tqdm(val_loader, desc=f'Validation Epoch {epoch + 1}', leave=False):
            if val_batches_processed >= max_val_batches:
                break
            x_baseline, pert, x_target_delta = batch
            x_baseline, pert, x_target_delta = x_baseline.to(
                device), pert.to(device), x_target_delta.to(device)
            x_target_abs = x_baseline + x_target_delta
            pred_noise, pert_pred, noise, t = model(
                x_target_abs, pert, is_train=False)
            diffusion_loss = F.mse_loss(pred_noise, noise)
            pert_loss = F.mse_loss(pert_pred, pert)
```

```
                loss = diffusion_loss + params['aux_weight'] * pert_loss
                val_loss += loss.item()
                val_batches_processed += 1

        val_loss /= val_batches_processed
        scheduler.step()

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
            torch.save(model.state_dict(),
                       f'best_model_trial_{trial.number}_{timestamp}.pt')
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f'Early stopping at epoch {epoch + 1}')
                break

        print(
            f'Epoch {epoch + 1}/{max_epochs} - Train Loss: {train_loss:.4f} - Val Loss: {val_loss:.4f}')

    return best_val_loss


def main(gpu_id=None):
    set_global_seed(42)
    global train_adata, test_adata, train_dataset, test_dataset, device, pca_model

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    print(f'Training started at: {timestamp}')

    if torch.cuda.is_available():
        gpu_count = torch.cuda.device_count()
        print(f'Available GPUs: {gpu_count}')
        if gpu_id is not None:
            if gpu_id >= gpu_count:
                print(
                    f'Warning: Specified GPU {gpu_id} not available, using GPU 0')
                gpu_id = 0
            device = torch.device(f'cuda:{gpu_id}')
            print(f'Using specified GPU {gpu_id}: {device}')
        else:
            device = torch.device('cuda:0')
            print(f'Using default GPU 0: {device}')
    else:
        device = torch.device('cpu')
        print('CUDA not available, using CPU')

    print('Loading ATAC data...')
    train_path = "datasets/LiscovitchBrauerSanjana2021_train_filtered2.h5ad"
    test_path = "datasets/LiscovitchBrauerSanjana2021_test_filtered2.h5ad"
    print(os.path.exists(train_path), os.path.exists(test_path))
    if not os.path.exists(train_path) or not os.path.exists(test_path):
        raise FileNotFoundError(
            f"Data files not found: {train_path} or {test_path}")

    train_adata = sc.read_h5ad(train_path)
    test_adata = sc.read_h5ad(test_path)
    print(f'Train data shape: {train_adata.shape}')
    print(f'Test data shape: {test_adata.shape}')

    print("Processing gene set consistency...")
    train_genes = set(train_adata.var_names)
    test_genes = set(test_adata.var_names)
    common_genes = list(train_genes & test_genes)
    print(f"Train genes: {len(train_genes)}")
    print(f"Test genes: {len(test_genes)}")
    print(f"Common genes: {len(common_genes)}")

    common_genes.sort()
    train_gene_idx = [train_adata.var_names.get_loc(
        gene) for gene in common_genes]
    test_gene_idx = [test_adata.var_names.get_loc(
        gene) for gene in common_genes]

    pca_model = None
    if scipy.sparse.issparse(train_adata.X):
        train_data = train_adata.X[:, train_gene_idx].toarray()
    else:
        train_data = train_adata.X[:, train_gene_idx]

    train_data = np.maximum(train_data, 0)
    train_data = np.maximum(train_data, 1e-10)
    train_data = np.log1p(train_data)
    scaler = StandardScaler()
    train_data = scaler.fit_transform(train_data)
    train_data = np.clip(train_data, -10, 10)
    train_data = train_data / 10.0

    common_genes_info = {
        'genes': common_genes,
        'train_idx': train_gene_idx,
        'test_idx': test_gene_idx
    }

    train_dataset = ATACDataset(
        train_adata,
        perturbation_key='perturbation',
        scaler=scaler,
        pca_model=None,
        pca_dim=128,
        fit_pca=False,
        augment=True,
        is_train=True,
        common_genes_info=common_genes_info
```

```
)
test_dataset = ATACDataset(
    test_adata,
    perturbation_key='perturbation',
    scaler=scaler,
    pca_model=None,
    pca_dim=128,
    fit_pca=False,
    augment=False,
    is_train=False,
    common_genes_info=common_genes_info
)

optuna.logging.set_verbosity(optuna.logging.WARNING)
study = optuna.create_study(
    direction='minimize',
    sampler=optuna.samplers.TPESampler(seed=42),
    pruner=optuna.pruners.MedianPruner(
        n_startup_trials=5,
        n_warmup_steps=5,
        interval_steps=1
    )
)

print('Starting hyperparameter optimization...')
study.optimize(lambda trial: objective(trial, timestamp), n_trials=50)

pert_dim, _ = standardize_perturbation_encoding(
    train_dataset, test_dataset)
best_params = study.best_params

n_genes = train_dataset.n_genes
print_log(
    f"Model input dim (full genes): {n_genes}, Model output dim (full genes): {n_genes}")

base_model = ConditionalDiffusionModel(
    input_dim=n_genes,
    output_dim=n_genes,
    train_pert_dim=pert_dim,
    test_pert_dim=pert_dim,
    hidden_dim=best_params['n_hidden'],
    n_layers=best_params['n_layers'],
    n_heads=best_params['n_heads'],
    dropout=best_params['dropout'],
    time_emb_dim=best_params['time_emb_dim'],
    pert_emb_dim=best_params['pert_emb_dim'],
    diffusion_steps=best_params['diffusion_steps']
)
final_model = DiffusionModel(
    base_model, diffusion_steps=best_params['diffusion_steps']).to(device)

train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(
    train_dataset, [train_size, val_size])

train_loader = DataLoader(
    train_subset,
    batch_size=best_params['batch_size'],
    shuffle=True,
    num_workers=4,
    pin_memory=True
)
val_loader = DataLoader(
    val_subset,
    batch_size=best_params['batch_size'],
    shuffle=False,
    num_workers=4,
    pin_memory=True
)
test_loader = DataLoader(
    test_dataset,
    batch_size=best_params['batch_size'],
    shuffle=False,
    num_workers=4,
    pin_memory=True
)

optimizer = torch.optim.AdamW(
    final_model.parameters(),
    lr=best_params['learning_rate'],
    weight_decay=best_params['weight_decay']
)
scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    optimizer,
    T_0=10,
    T_mult=2,
    eta_min=1e-6
)

print('Training final model...')
best_loss = float('inf')
best_model_state = None
max_epochs = 300
patience = 25
patience_counter = 0

for epoch in range(max_epochs):
    train_loss = train_model(final_model, train_loader, optimizer, scheduler, device,
                             aux_weight=best_params['aux_weight'])
    val_metrics = evaluate_model(final_model, val_loader, device,
                                 aux_weight=best_params['aux_weight'])

    if (epoch + 1) % 30 == 0:
        print(f'Epoch {epoch + 1}/{max_epochs}:')
```

```
            print(f'Training Loss: {train_loss:.4f}')
            print(f'Validation Loss: {val_metrics["loss"]:.4f}')

        if val_metrics["loss"] < best_loss:
            best_loss = val_metrics["loss"]
            patience_counter = 0
            # [OK] deep copy to avoid being overwritten by later updates
            best_model_state = copy.deepcopy(final_model.state_dict())
            torch.save({
                'epoch': epoch,
                'model_state_dict': best_model_state,
                'optimizer_state_dict': optimizer.state_dict(),
                'scheduler_state_dict': scheduler.state_dict(),
                'loss': best_loss,
                'metrics': val_metrics,
                'best_params': best_params
            }, f'atac_diffusion_best_model_{timestamp}.pt')
            print(f"Saved best model with validation loss: {best_loss:.4f}")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f'Early stopping at epoch {epoch + 1}')
                break

    if best_model_state is not None:
        final_model.load_state_dict(best_model_state)
    print('Evaluating final model on test set...')
    results = evaluate_and_save_model(final_model, test_loader, device,
                                      f'atac_diffusion_final_model_{timestamp}.pt',
                                      common_genes_info, pca_model, scaler, best_params)

    best_params_str = str(best_params)
    if len(best_params_str) > 50:
        best_params_str = best_params_str[:50] + '...'

    results_df = pd.DataFrame({
        'Metric': ['MSE', 'PCC', 'R2', 'MSE_DE', 'PCC_DE', 'R2_DE'],
        'Value': [results['MSE'], results['PCC'], results['R2'],
                  results['MSE_DE'], results['PCC_DE'], results['R2_DE']],
        'Best_Params': [best_params_str] * 6
    })
    results_df.to_csv(
        f'atac_diffusion_evaluation_results_{timestamp}.csv', index=False)
    print_log("\nFinal Evaluation Results:")
    print_log(results_df.to_string(
        index=False, float_format=lambda x: '{:.6f}'.format(x)))
    return results_df


def load_model_for_prediction(model_path, device=None):
    if device is None:
        device = 'cuda' if torch.cuda.is_available() else 'cpu'

    print(f"Loading model from {model_path}")
    # [OK] more compatible torch.load
    checkpoint = torch.load(model_path, map_location=device)

    if 'model_state_dict' not in checkpoint or 'model_config' not in checkpoint:
        raise ValueError(
            f"Checkpoint at {model_path} does not contain 'model_state_dict' or 'model_config'. "
            f"Please pass a final evaluation checkpoint (atac_diffusion_final_model_*.pt)."
        )

    model_state = checkpoint['model_state_dict']
    predictions = checkpoint.get('predictions', None)
    targets = checkpoint.get('targets', None)
    gene_names = checkpoint.get('gene_names', None)
    pca_model = checkpoint.get('pca_model', None)
    scaler = checkpoint.get('scaler', None)
    model_config = checkpoint['model_config']
    evaluation_results = checkpoint.get('evaluation_results', None)

    output_dim = model_config.get('output_dim', model_config['input_dim'])

    base_model = ConditionalDiffusionModel(
        input_dim=model_config['input_dim'],
        output_dim=output_dim,
        train_pert_dim=model_config['train_pert_dim'],
        test_pert_dim=model_config['test_pert_dim'],
        hidden_dim=model_config['hidden_dim'],
        n_layers=model_config['n_layers'],
        n_heads=model_config['n_heads'],
        dropout=model_config['dropout'],
        time_emb_dim=model_config.get('time_emb_dim', 128),
        pert_emb_dim=model_config.get('pert_emb_dim', 64),
        diffusion_steps=model_config.get('diffusion_steps', 1000),
        model_in_dim=model_config.get('model_in_dim', 128),
        bottleneck_dims=tuple(model_config.get(
            'bottleneck_dims', (2048, 512))),
        use_bottleneck=model_config.get('use_bottleneck', True)
    )
    model = DiffusionModel(base_model, diffusion_steps=model_config.get(
        'diffusion_steps', 1000)).to(device)

    model.load_state_dict(model_state)
    model.eval()

    print(f"Model loaded successfully!")
    print(
        f"Gene names: {len(gene_names) if gene_names is not None else 'None'}")
    if predictions is not None:
        print(f"Predictions shape: {predictions.shape}")
    if targets is not None:
        print(f"Targets shape: {targets.shape}")
```

```
    return {
        'model': model,
        'predictions': predictions,
        'targets': targets,
        'gene_names': gene_names,
        'pca_model': pca_model,
        'scaler': scaler,
        'model_config': model_config,
        'evaluation_results': evaluation_results
    }


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='ATAC Conditional Diffusion Model Training'
    )
    parser.add_argument('--gpu', type=int, default=None,
                        help='Specify GPU number to use (e.g., --gpu 0 for GPU 0, --gpu 1 for GPU 1)')
    parser.add_argument('--list-gpus', action='store_true',
                        help='List all available GPUs and exit')
    args = parser.parse_args()

    if args.list_gpus:
        if torch.cuda.is_available():
            gpu_count = torch.cuda.device_count()
            print(f'Available GPUs: {gpu_count}')
            for i in range(gpu_count):
                print(f'GPU {i}: {torch.cuda.get_device_name(i)}')
        else:
            print('CUDA not available')
        exit(0)

    print("=" * 60)
    print("ATAC Conditional Diffusion Model Training")
    print("=" * 60)

    if args.gpu is not None:
        print(f"Using specified GPU: {args.gpu}")
    else:
        print("Using default GPU settings")

    results_df = main(gpu_id=args.gpu)
```