Supplementary Information for Learning efficient representations of complex constraints for scalable optimization

Contents

1 How to use PolyFormer

1.1 Overview

PolyFormer is implemented in a Python environment. To run PolyFormer, you will need to install the following packages:

- `pyomo` package (for modelling optimization problems),
- `pytorch` package (for training execution),
- Necessary optimization solvers (for convex problems, we recommend Cplex and Gurobi; for nonconvex problems, we recommend IPOPT. Of course, other solvers that can be called by `pyomo` are also acceptable).

The main algorithm of PolyFormer is packaged into a Python file named `Approximator.py`. The key components in `Approximator.py` that users need to call/modify are as follows:

- `class ErrorCalculator()` stores the model and calculates the directional errors.
- `class PreTrainNet(nn.Module)`, `class BiasNet(nn.Module)`, and `class FullNet(nn.Module)` define various types of neural networks.
  - `PreTrainNet` is a single-layer neural network with no inputs and cannot handle situations involving the varying parameter $\boldsymbol{\theta}$. It is used to identify the shape of the approximate polytope and to initialize the model with a warm start, accelerating the training process when $\boldsymbol{\theta}$ is introduced.
  - `BiasNet` only fits the mapping relationship $\mathbf{b}(\boldsymbol{\theta})$, keeping the matrix $\mathbf{A}$ fixed.
  - `FullNet` fits the mapping relationships $\mathbf{A}(\boldsymbol{\theta})$ and $\mathbf{b}(\boldsymbol{\theta})$.

  In our implementation of PolyFormer, both `FullNet` and `BiasNet` use neural networks with a single hidden layer, and the hyperparameter `n_hidden` controls the size of this hidden layer. Users of PolyFormer can modify these networks or construct new networks based on prior knowledge of their specific application scenarios to improve performance in those contexts.
- `class Trainer()` executes the training process.

The basic workflow is that we define an initial feasible region model externally, and then call the functions/classes within `Approximator.py` to execute training. Therefore, the main area where users need to program is in defining the original model. These mainly include the following aspects:

1. Building the original `pyomo` model.
2. Defining the dataset for $\boldsymbol{\theta}$, which determines the source of sampling $\boldsymbol{\theta}$ during the PolyFormer training process.
3. Defining training callbacks, which are typically used for outputting intermediate results and monitoring the training process.
4. Defining hyperparameters, such as learning rates, weight coefficients, optimizers, etc.

In the next section, a simple step-by-step example will be provided to demonstrate the usage of PolyFormer.

1.2 A step-by-step example

1.2.1 Defining the case

As an example, let us consider the Octagon case which is detailed in Supplementary Note 2. The code that defines this case is as follows:

Octagon case definition

```python
import numpy as np
from Simulator.Approximator import ErrorCalculator, pyomo_params_to_numpy
import torch
from torch.utils.data import Dataset, DataLoader
from pyomo.environ import *
from Simulator import PROJECT_ROOT

def case_polygon(total_samples=200, noise_scale=0.10, batch_size=1, model_type='pretrainnet', device='cpu'):
    """Implementation of original case with fixed parameters"""
    # Fixed initialization parameters
    A_init = np.array([
        [1, 0], [0, 1], [-1, 0], [0, -1],
        [1, 1], [-1, -1], [1, -1], [-1, 1]
    ])
    b_init = np.array([1, 1, 0, 0, 1.5, -0.5, 0.7, 0.7])

    # Automatically deduce dimensions
    dim = A_init.shape[1]
    ncons = A_init.shape[0]
    dim_theta = 2
    num_b = ncons - dim_theta

    # Build model
    model = ConcreteModel()

    # Define parameters
    model.A = Param(range(ncons), range(dim),
                    initialize={(i, j): A_init[i, j] for i, j in np.ndindex(A_init.shape)},
                    mutable=True)
    model.b = Param(range(num_b),
                    initialize={i: b_init[i] for i in range(num_b)},
                    mutable=True)
    model.theta = Param(range(dim_theta),
                        initialize={i: b_init[num_b + i] for i in range(2)},
                        mutable=True)

    # Define variables
    model.var_proj = Var(range(dim), domain=Reals)

    # Define constraints
    def constraint_rule(model, i, is_adjustable):
        idx = i + num_b if is_adjustable else i
        param = model.theta[i] if is_adjustable else model.b[i]
        return sum(model.A[idx, j] * model.var_proj[j] for j in range(dim)) <= param

    model.con_fixed = Constraint(range(num_b), rule=lambda m, i: constraint_rule(m, i, False))
    model.con_adj = Constraint(range(dim_theta), rule=lambda m, i: constraint_rule(m, i, True))
```

```
48
49      original_model = {'model': model}
50
51      # Dataset configuration
52      class CaseData(Dataset):
53          def __init__(self, size=total_samples):
54              self.size = size
55              self.noise_scale = noise_scale
56
57          def __len__(self):
58              return self.size
59
60          def __getitem__(self, idx):
61              return {'theta': torch.normal(0, self.noise_scale, (dim_theta,), device=
    ↪ device)}
62
63      # Initialize approximation
64      A_hat = np.vstack([
65          np.eye(dim),   # Upper bound
66          -np.eye(dim),  # Lower bound
67      ])
68
69      errorcalculator = ErrorCalculator(
70          original_model=original_model,
71          A_hat=A_hat,
72          solver='cplex',
73      )
74
75      # Define training callback function (user customization)
76      def callback(errorcalculator, epoch):
77          # Print training metrics
78          len_his = len(errorcalculator.training_history['feas'])
79          print(f"Iter{epoch}:FeasErr={np.mean(errorcalculator.training_history['feas
    ↪ '][-min(10,len_his):]):.2e}, "
80                f"OptErr={np.mean(errorcalculator.training_history['opt'][-min(10,
    ↪ len_his):]):.2e}")
81
82          # Save intermediate results
83          if epoch % 100 == 0:
84              # Custom save logic
85              pass
86
87      # Trainer configuration
88      if model_type.lower() == 'pretrainnet':
89          trainer_configure = {
90              'call_interval': 20,
91              'training_callback': callback,
92              "optimizer": "SGD",
93              "lr": 0.4,
94              "batch_size": 1,
95              "scheduler": {
96                  "type": "StepLR",
97                  "step_size": 100,
98                  "gamma": 0.98
99              },
100             "n_cal": 2,
101             "cal_feas": True,
102             "cal_opt": True,
103             "rate_opt_feas": 1
104         }
```

```
105     else:
106         trainer_configure = {
107             "call_interval": 1,
108             "training_callback": callback,
109             "optimizer": "adam",
110             "lr": 0.01,
111             "batch_size": batch_size,
112             "scheduler": {
113                 "type": "StepLR",
114                 "step_size": 100,
115                 "gamma": 0.99
116             },
117             "n_cal": 3,
118             "cal_feas": True,
119             "cal_opt": True,
120             "rate_opt_feas": 1,
121         }
122 
123     params_dict, param_count = pyomo_params_to_numpy(model)
124     params = {
125         'params_dict': params_dict,
126         'dataloader': DataLoader(CaseData(), batch_size=batch_size, shuffle=True),
127         'count': dim_theta,
128     }
129 
130     case_name = 'polygon'
131     return {
132         'casename': case_name,
133         'A_hat': errorcalculator.A_hat,
134         'b_hat': errorcalculator.b_hat,
135         'errorcalculator': errorcalculator,
136         'params': params,
137         'trainer_configure': trainer_configure,
138         'result_path': f'{PROJECT_ROOT}/results/{case_name}/{model_type.lower()}
    ↪ _weights.pth',
139         'metadata': {'A_init': A_init, 'b_init': b_init}
140     }
```

The above code defines the Octagon case with varying parameters. The main parts of this code are as follows:

Part 0: Imports (lines 1-7)

This part imports the necessary packages and modules required for defining the model and executing PolyFormer, i.e., `numpy`, `pyomo`, `torch`, and the relevant classes from `Approximator.py`.

Part 1: Building the Original Model (lines 9-49)

This part constructs the original optimization model using `pyomo`. It defines the parameters, variables, and constraints of the model. The parameters include the matrix $\mathbf{A}$, the fixed vector $\mathbf{b}$, and the adjustable parameter vector $\boldsymbol{\theta}$. We set the last two elements of $\mathbf{b}$ as $\boldsymbol{\theta}$. The constraints are defined based on these parameters.

The variable `var_proj` (defined in line 38) represents the variables of interest for which we aim to construct the feasible region, i.e., the $\mathbf{x}$ in Equation (1) of the main text. We recommend PolyFormer users not to modify the name `var_proj` in any case to ensure consistency and uniformity of the code. If users employ different variable

names in their own applications, they can simply map those variables to `var_proj` via an equality constraint.

The constraints are defined in two parts: fixed constraints (line 46) and adjustable constraints (line 47). The fixed constraints use the fixed vector $\mathbf{b}$, while the adjustable constraints use the parameter vector $\boldsymbol{\theta}$.

When the original model is defined, we store it in a dictionary named `original_model` (line 49) for later use in PolyFormer.

Part 2: Dataset Configuration (lines 51-64)

This component defines a custom dataset class, `CaseData`, which generates samples of the parameter $\boldsymbol{\theta}$ with additive Gaussian noise. The dataset is used during training to draw diverse realizations of $\boldsymbol{\theta}$. The `__getitem__` method (lines 60-62) returns a single sample as a dictionary containing a PyTorch tensor that encodes the perturbation of $\boldsymbol{\theta}$ from its initial value. Consequently, the $\mathbf{A}$-net and $\mathbf{b}$-net learn a mapping from these perturbations to approximate polytopes, rather than from absolute parameter values. This design enhances training performance, particularly when the parameter perturbations are small.

In this case, the dictionary key is "`theta`" because the parameter defined in the original Pyomo model is named `theta`. In general, the dictionary key does not need to be `theta`, but it must be identical to the name of the varying parameter defined in the original model to ensure compatibility with PolyFormer. For example, if the varying parameter in the original model is named `param1`, then the dictionary returned by `__getitem__` should be `{'param1': ...}`. The dimension of the sampled parameter should also be consistent with the definition in the original model. If a case contains multiple varying parameters, such as `param1` and `param2`, then the returned dictionary should be `{'param1': ..., 'param2': ...}`.

Part 3: Initializing the Approximator (lines 63-73)

This part initializes the `ErrorCalculator` class with the original model and an initial approximation of the feasible region. We only need to initialize the $\mathbf{A}$ matrix that defines the approximate polytope, while the $\mathbf{b}$ vector will be automatically initialized within the `ErrorCalculator` class based on the original model.

Of course, the `ErrorCalculator` class also allows users to manually specify $\mathbf{b}$ by adding an input argument `b_hat`. This is useful when users have prior knowledge about the feasible region, which avoids the initialization step and can also speed up the training process.

The user also needs to specify the solver used to solve the optimization problems during the training process. Here, the designation `solver = 'cplex'` refers to the solver employed to handle the optimization problems involving the original feasible region. On the other hand, the optimization problems defined over the approximated polytope are solved using the solver specified by the parameter `cvx_solver` (as optimization on the approximated polytope in PolyFormer is always convex). The default value of `cvx_solver` has already been set to `'cplex'` in the `ErrorCalculator` class. Users can choose other solvers supported by `pyomo` as needed.

Part 4: Defining the Training Callback Function (lines 75-85)

This part defines a callback function that is called during the training process to monitor progress and output intermediate results. In this example, the callback function prints the average feasibility and optimality errors over the last 10 iterations. Users can customize this function based on their specific needs.

Part 5: Trainer Configuration (lines 87-121)

This part sets up the configuration for PolyFormer training. Different hyperparameters are specified depending on the type of model being trained (`pretrainnet` or others). The hyperparameters include:

- `call_interval`: The interval at which the callback function is called during training.
- `training_callback`: The callback function defined in Part 4.
- `optimizer`: The optimization algorithm used for training (e.g., SGD, Adam).
- `lr`: The learning rate for the optimizer. The user can also set different learning rates for $\mathbf{A}$ and $\mathbf{b}$ by setting `lr_A` and `lr_b`, respectively.
- `batch_size`: The batch size for the samples of $\boldsymbol{\theta}$ during training.
- `scheduler`: A learning rate scheduler to adjust the learning rate during training.
- `n_cal`: The number of sampled directions used for directional error calculation in each training iteration.
- `cal_feas` and `cal_opt`: Flags indicating whether to calculate feasibility and optimality errors, respectively.
- `rate_opt_feas`: The weight ratio between optimality and feasibility errors in the loss function: 1 means the feasibility and optimality errors are equally weighted; 0 means only the feasibility error is considered.

These configurations are stored in the `trainer_configure` dictionary for later use during training.

Part 6: Returning Case Information (lines 123-140)

Finally, the function returns a dictionary containing all the necessary information for training PolyFormer in this case. This includes the `casename`, initial $\mathbf{A}$ and $\mathbf{b}$, `A_hat` and `b_hat`, the `ErrorCalculator` instance, parameter settings `params` (including all the names of varying parameters and their initial values as the dictionary `params_dict`, the dataset `dataloader` for sampling varying parameters, and the total number of varying parameters `count`), `trainer_configuration`, the `result_path` to save the trained model weights, and any `metadata` of the original model that users may want to store.

1.2.2 Train the PolyFormer

After defining the case as shown above, we can proceed to train PolyFormer. The training code is as follows:

Training script

```
import os
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'
from Simulator.Approximator import PreTrainNet,BiasNet,FullNet,compute_loss,Trainer
import torch
```

```
from Simulator.cases.basic_cases import case_polygon
from Simulator import PROJECT_ROOT
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
parallel = True
model_type = 'fullnet'
case = case_polygon(model_type = model_type,device=device)
if model_type=='pretrainnet':
    n_train = 1000
    model   = PreTrainNet(case['A_hat'],case['b_hat'],device = device).to(device)
else:
    n_train = 20
    model = PreTrainNet(case['A_hat'], case['b_hat'])
    model.load_state_dict(torch.load(f'{PROJECT_ROOT}\\results\\{case['casename']}\\
    ↪ pretrainnet_weights.pth', map_location=device))
    A_pretrained, b_pretrained = model()
    b_pretrained = b_pretrained[0].detach().cpu().numpy()
    if model_type == 'biasnet':
        A_pretrained = A_pretrained[0].detach().to(device)
        case['trainer_configure'].update(A_pretrained = A_pretrained)
        model = BiasNet(dim_theta=case['params']['count'], b_init=b_pretrained,device
    ↪ = device).to(device)
    elif model_type == 'fullnet':
        A_pretrained = A_pretrained[0].detach().cpu().numpy()
        model= FullNet(dim_theta = case['params']['count'], A_init=A_pretrained,b_init
    ↪  = b_pretrained,device = device).to(device)
trainer = Trainer(
    model=model,
    error_calculator=case['errorcalculator'],
    compute_loss=compute_loss,
)
trainer.configure(**case['trainer_configure'])
trainer.initialize()
trainer.train(n_train = n_train , params_data=case["params"], parallel = parallel)
torch.save(model.state_dict(), case['result_path'])
```

In the training code, first (lines 1-10), we import a series of necessary packages and define the `case`, `model_type`, and the parallel flag `parallel`. When `parallel` is set to `False`, samples in a batch are solved sequentially. When `parallel` is set to `True`, all samples in a batch are solved in parallel using multi-threading on the CPU, accelerating the training process. This works only if the solver supports parallelism, e.g., `cplex`. Additionally, both the **A**-net and **b**-net are trained on the GPU, but the loss functions are calculated using CPU-based optimization solvers, as mature GPU-based solvers are not yet available. Consequently, the program frequently transfers data between the CPU and GPU. In the future, as GPU-based optimization techniques mature, all computations could be deployed on the GPU, leading to substantial efficiency improvements.

Next, in lines 11-26, we initialize different types of models. The `pretrainnet` is initialized as the external approximation of the original region (as explained in Remark 1 of the main text). Both `biasnet` and `fullnet` are initialized based on the output of `pretrainnet`. Besides, different training iteration counts are set for each model type. In this case, `pretrainnet` is trained for 1000 iterations, while the other networks are trained for 20 iterations. This difference arises because `pretrainnet` does not have a dataset, so 1000 iterations represent 1000 updates. In contrast, the other networks have datasets, and the 20 iterations correspond to 20 passes through the dataset. Given

that the parameter dataset consists of 200 samples (as defined in `case_polygon`), the total number of iterations for the `biasnet` and `fullnet` is $20 \times 200 = 4000$.

Finally, in lines 27-35, all arguments are applied to the trainer, the training process is executed, and the results are saved.

1.2.3 Using a trained PolyFormer

After training, PolyFormer can be used to quickly fit the feasible region under various parameter values. To this end, we only need to load the trained model weights and input the new parameter values to obtain an approximate polytope. The resulting approximate polytope can subsequently be employed for a variety of downstream tasks, such as rapid feasibility checking and optimization, which we do not elaborate on further here.

2 Numerical settings of typical geometries

We provide here the numerical configurations of the five representative geometries discussed in the main text—including polygons, ellipses, nonconvex regions, hypercubes, and balls—to facilitate implementation of PolyFormer. Although the PolyFormer project provides three model variants—`PreTrainNet`, `BiasNet`, and `FullNet`—all numerical experiments shown in the main text employ only `PreTrainNet` and `FullNet`. Consequently, the numerical configurations below list only the training hyperparameters used for `PreTrainNet` and `FullNet`.

2.1 Polygon

We consider an octagon defined by the following inequalities:

$$\begin{aligned} 0 \le x_1 &\le 1 \\ 0 \le x_2 &\le 1 \\ 0.5 \le x_1 + x_2 &\le 1.5 \\ -\theta_1 \le x_1 - x_2 &\le \theta_2 \end{aligned} \tag{S.1}$$

where, $\mathbf{x} = (x_1, x_2)$ are the decision variables, and $\boldsymbol{\theta} = (\theta_1, \theta_2)$ are the varying parameters. The initial values of $\boldsymbol{\theta}$ are set to $(0.7, 0.7)$, respectively. The dataset of $\boldsymbol{\theta}$ includes 200 samples, derived from a normal distribution with a mean of $(0.7, 0.7)$ and a standard deviation of 0.1 in both dimensions. The number of rows M in the approximate polytope is set to 4. The initial $\mathbf{A}$ matrix defining the approximate polytope is set as:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \tag{S.2}$$

The training hyperparameters for `PreTrainNet` and `FullNet` are listed in Table 1.

Table 1 Training hyperparameters in the polygon case

Model types	`PreTrainNet`	`FullNet`
`n_train`	1000	20
`optimizer`	SGD	Adam
`lr`	0.4	0.01
`solver`	`cplex`	`cplex`
`cvx_solver`	`cplex`	`cplex`
`n_hidden`	0	64
`batch_size`	1	1
`n_cal`	2	3
`cal_feas`	True	True
`cal_opt`	True	True
`rate_opt_feas`	1	1
`scheduler`	StepLR (step size=100, gamma=0.98)	StepLR (step size=100, gamma=0.99)

2.2 Ellipse

The ellipse is defined by the following quadratic constraint:

$$\theta_1 x_1^2 + 2\theta_2 x_1 x_2 + \theta_1 x_2^2 \leq 1 \tag{S.3}$$

where $\mathbf{x} = (x_1, x_2)$ are the decision variables, and $\boldsymbol{\theta} = (\theta_1, \theta_2)$ are the varying parameters. The initial values of $\boldsymbol{\theta}$ are set to $(2.5, -1.5)$, respectively. The dataset of $\boldsymbol{\theta}$ includes 100 samples, derived from a normal distribution with a mean of $(2.5, -1.5)$ and a standard deviation of 0.2 in both dimensions. The number of rows M in the approximate polytope is set to 6. The initial $\mathbf{A}$ matrix defining the approximate polytope is set as:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix} \tag{S.4}$$

The training hyperparameters for `PreTrainNet` and `FullNet` are listed in Table 2.

2.3 Nonconvex region

The nonconvex region is defined by the following constraints:

$$\begin{aligned} x_1^2 + x_2^2 &\leq 1 \\ (x_1 - \theta_1)^2 + (x_2 - \theta_2)^2 &\geq \theta_3^2 \end{aligned} \tag{S.5}$$

where $\mathbf{x} = (x_1, x_2)$ are the decision variables with bounds $x_1, x_2 \in [-1, 1]$, and $\boldsymbol{\theta} = (\theta_1, \theta_2, \theta_3)$ are the varying parameters representing the centre coordinates and radius of the exclusion circle. The initial values of $\boldsymbol{\theta}$ are set to $(1, 1, 1)$, respectively. The

Table 2 Training hyperparameters in the ellipse case

Model types	`PreTrainNet`	`FullNet`
`n_train`	300	15
`optimizer`	SGD	Adam
`lr`	0.2	0.002
`solver`	`cplex`	`cplex`
`cvx_solver`	`cplex`	`cplex`
`n_hidden`	0	64
`batch_size`	1	5
`n_cal`	2	2
`cal_feas`	True	True
`cal_opt`	True	True
`rate_opt_feas`	1	1
`scheduler`	StepLR (step size=100, gamma=0.95)	StepLR (step size=100, gamma=0.95)

dataset of $\boldsymbol{\theta}$ includes 200 samples, derived from a normal distribution with a mean of $(1, 1, 1)$ and a standard deviation of 0.3 in all dimensions. The number of rows M in the approximate polytope is set to 6. The initial $\mathbf{A}$ matrix defining the approximate polytope is set as:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix} \tag{S.6}$$

The training hyperparameters for `PreTrainNet` and `FullNet` are listed in Table 3. Note that due to the nonconvex nature of the problem, the solver is set to `ipopt` to handle the nonconvex constraints.

Table 3 Training hyperparameters in the nonconvex case

Model types	`PreTrainNet`	`FullNet`
`n_train`	300	20
`optimizer`	SGD	Adam
`lr`	0.2	0.006
`solver`	`ipopt`	`ipopt`
`cvx_solver`	`cplex`	`cplex`
`n_hidden`	0	64
`batch_size`	5	5
`n_cal`	2	5
`cal_feas`	True	True
`cal_opt`	True	True
`rate_opt_feas`	1	1
`scheduler`	StepLR (step size=100, gamma=0.99)	StepLR (step size=100, gamma=0.95)

2.4 Hypercube

In this case, the original feasible region is a d-dimensional hypercube, which is defined by the following constraints:

$$-1 \leq x_i \leq 1, \quad \forall i = 1, 2, \ldots, d \tag{S.7}$$

where, $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ are the decision variables and d varies over the set $\{2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 140, 160, 180, 200\}$.

The hypercube case focuses on scalability analysis across different dimensions. It does not involve any varying parameters $\boldsymbol{\theta}$, and thus the dataset for $\boldsymbol{\theta}$ is set to `None`. Consequently, only `PreTrainNet` is applicable. The number of rows M in the approximate polytope is set to $2d$.

The initial $\mathbf{A}$ matrix defining the approximate polytope is constructed as:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} \mathbf{I}_d \\ -\mathbf{I}_d \end{bmatrix} + \mathbf{E} \tag{S.8}$$

where $\mathbf{I}_d$ is the d-dimensional identity matrix, and $\mathbf{E} \in \mathbb{R}^{2d \times d}$ is a noise matrix in which each element e_{ij} is independently drawn from the Gaussian distribution $\mathcal{N}(0, (\sqrt{\frac{1}{2d}})^2)$.

The training process employs a two-phase strategy with different dimension-adaptive training iterations and learning rates:

- Phase 1: `n_train`= `int`$(50\sqrt{d/2})$ epochs with learning rate `lr` $= 0.02\sqrt{2/d}$
- Phase 2: `n_train`= `int`$(400\sqrt{d/2})$ epochs with learning rate `lr` $= 0.3\sqrt{2/d}$

Other training hyperparameters are listed in Table 4.

Table 4 Training hyperparameters in the hypercube case

Hyperparameter	Value
`optimizer`	SGD
`solver`	`gurobi`
`cvx_solver`	`gurobi`
`n_cal`	3
`cal_feas`	True
`cal_opt`	True
`rate_opt_feas`	1
`scheduler`	`StepLR` (step size=100, gamma=1.05)

2.5 Unit ball

In this case, the original feasible region is a d-dimensional unit ball defined by the following quadratic constraint:

$$\sum_{i=1}^{d} x_i^2 \leq 1 \tag{S.9}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ are the decision variables, and d varies over the set $\{2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 140, 160, 180, 200\}$. Similar to the hypercube case, this case focuses on scalability analysis across different dimensions, so we do not test the `PreTrainNet` (without any varying parameters). The number of rows M in the approximate polytope is set to $2d$. The initial $\mathbf{A}$ matrix defining the approximate polytope is constructed identically as in Equation (S.8).

The training process also employs a two-phase strategy with different training iterations and learning rates:

- Phase 1: `n_train`= `int`$(100\sqrt{d/2})$ epochs with learning rate `lr` $= 0.01\sqrt{2/d}$;
- Phase 2: `n_train`= `int`$(500\sqrt{d/2})$ epochs with learning rate `lr` $= 0.2\sqrt{2/d}$.

Other training hyperparameters are listed in Table 5.

Table 5 Training hyperparameters in the unit-ball case

Hyperparameter	Value
`optimizer`	SGD
`solver`	`gurobi`
`cvx_solver`	`gurobi`
`n_cal`	2
`cal_feas`	True
`cal_opt`	True
`rate_opt_feas`	1
`scheduler`	None

3 Numerical settings of resource aggregation

3.1 Scenario setup

We consider the aggregation of three types of devices, i.e., electric vehicles (EVs), battery storage systems (BSSs), and heat pumps (HPs), over a time horizon of $T = 24$ hours with a time resolution of $\Delta_t = 1$ hour. Two scenarios with different device portfolios are studied:

- **Scenario 1**: 600 EVs and 400 HPs, all continuously controlled.
- **Scenario 2**: 54 continuously-controlled EVs, 36 continuously-controlled HPs, 6 on-off controlled EVs, 4 on-off controlled HPs, and 5 BSSs.

3.2 Individual modelling of different resource types

The individual operational constraints for each device type are as follows:

EVs:

$$e_{i,t+1} = e_{i,t} + \eta_i p_{i,t} \Delta_t, \quad t \in [1, T-1], \tag{S.10a}$$

$$e_i^{\min} \le e_{i,t} \le e_i^{\max}, \quad t \in [1, T], \tag{S.10b}$$

$$e_{i,t_i^{\mathrm{d}}} \ge e_i^{\mathrm{req}}, \quad e_{i,t_i^{\mathrm{a}}} = e_i^{\mathrm{init}}, \tag{S.10c}$$

$$p_{i,t} = 0, \quad t \notin [t_i^{\mathrm{a}}, t_i^{\mathrm{d}}], \tag{S.10d}$$

if continuously controlled:

$$0 \le p_{i,t} \le p_i^{\max}, \quad t \in [t_i^{\mathrm{a}}, t_i^{\mathrm{d}}], \tag{S.10e}$$

if on-off controlled:

$$p_{i,t} = u_{i,t} p_i^{\max}, \quad u_{i,t} \in \{0, 1\}, \quad t \in [t_i^{\mathrm{a}}, t_i^{\mathrm{d}}], \tag{S.10f}$$

where the notation $[a, b]$ denotes the set of integers from a to b; $e_{i,t}$ is the state of charge (SoC) of EV i at time t; $p_{i,t}$ is the charging power; η_i is the charging efficiency; $e_i^{\min}$ and $e_i^{\max}$ are the minimum and maximum SoC limits; t_i^a and t_i^d are the arrival and departure times; e_i^{req} is the required SoC at departure; e_i^{init} is the initial SoC upon arrival; $p_i^{\max}$ is the maximum charging power; and $u_{i,t}$ is a binary variable indicating whether the EV is charging.

HPs:

$$q_{i,t+1} = \alpha_i q_{i,t} + (1 - \alpha_i)(q_t^{\mathrm{amb}} + \eta_i R_i p_{i,t}), \forall t \in [1, T-1] \tag{S.11a}$$

$$q_i^{\min} \le q_{i,t} \le q_i^{\max}, \forall t \in [1, T] \tag{S.11b}$$

$$q_{i,1} = q_i^{\mathrm{init}}, \tag{S.11c}$$

if continuously controlled:

$$0 \le p_{i,t} \le p_i^{\max}, \forall t \in [1, T], \tag{S.11d}$$

if on-off controlled:

$$p_{i,t} = u_{i,t} p_i^{\max}, u_{i,t} \in \{0, 1\}, \forall t \in [1, T], \tag{S.11e}$$

where $q_{i,t}$ is the indoor temperature of HP i at time t; α_i is the thermal retention coefficient, defined as $\alpha_i = e^{-\Delta_t/(R_i C_i)}$ with C_i being the thermal capacitance and R_i the thermal resistance; q_t^{amb} is the ambient temperature at time t; η_i is the coefficient of performance; $p_{i,t}$ is the heating power; $q_i^{\min}$ and $q_i^{\max}$ are the minimum and maximum temperature limits; q_i^{init} is the initial indoor temperature; $p_i^{\max}$ is the maximum heating power; and $u_{i,t}$ is a binary variable indicating whether the HP is on.

BSSs:

$$e_{i,t+1} = e_{i,t} + \eta_i^{\mathrm{chg}} p_{i,t}^{\mathrm{chg}} \Delta_t - \frac{1}{\eta_i^{\mathrm{dis}}} p_{i,t}^{\mathrm{dis}} \Delta_t, \quad \forall t \in [1, T-1], \tag{S.12a}$$

$$e_i^{\min} \le e_{i,t} \le e_i^{\max}, \quad \forall t \in [1, T], \tag{S.12b}$$

$$0 \le p_{i,t}^{\text{chg}} \le p_i^{\max} u_{i,t}^{\text{chg}}, \quad \forall t \in [1,T], \tag{S.12c}$$

$$0 \le p_{i,t}^{\text{dis}} \le p_i^{\max}(1 - u_{i,t}^{\text{chg}}), \quad \forall t \in [1,T], \tag{S.12d}$$

$$u_{i,t}^{\text{chg}} \in \{0,1\}, \quad \forall t \in [1,T], \tag{S.12e}$$

$$e_{i,1} = e_i^{\text{init}}, e_{i,T} \ge e_i^{\text{init}}, \tag{S.12f}$$

$$p_{i,t} = p_{i,t}^{\text{chg}} - p_{i,t}^{\text{dis}}, \quad \forall t \in [1,T], \tag{S.12g}$$

where $e_{i,t}$ is the SoC of BSS i at time t; $p_{i,t}^{\text{chg}}$ and $p_{i,t}^{\text{dis}}$ are the charging and discharging power; $p_{i,t}$ is the net power (positive for charging, negative for discharging); η_i^{chg} and η_i^{dis} are the charging and discharging efficiencies; $e_i^{\min}$ and $e_i^{\max}$ are the minimum and maximum SoC limits; $p_i^{\max}$ is the maximum charging/discharging power; $u_{i,t}^{\text{chg}}$ is a binary variable indicating whether the BSS is charging; and e_i^{init} is the initial SoC.

3.3 Resource parameters

EV parameters

Each EV is characterized by the following parameters:

$$\begin{aligned}
p^{\max} &= 7 \text{ kW} \quad \text{(rated charging power)} \\
B &= 50 \text{ kWh} \quad \text{(battery capacity)} \\
\eta_{\text{chg}} &= 0.95 \quad \text{(charging efficiency)} \\
\text{SOC}_{\text{a}} &\sim \mathcal{N}(0.5, 0.2^2), \text{ truncated to } [0.2, 0.8] \quad \text{(initial SOC)} \\
\text{SOC}_{\text{d}} &= 1 - P_{\text{chg}}/B \approx 0.86 \quad \text{(departure SOC)} \\
e^{\max} &= B \quad \text{(maximum energy)} \\
e^{\min} &= 0 \quad \text{(minimum energy)} \\
e^{\text{init}} &= \text{SOC}_{\text{a}} \times B \quad \text{(initial energy)} \\
e^{\text{req}} &= \text{SOC}_{\text{d}} \times B \quad \text{(required energy)} \\
t^{\text{a}} &\sim \mathcal{N}(8.5, 1.5^2), \text{ truncated to } [7, 10] \quad \text{(arrival time)} \\
t^{\text{d}} &= t^{\text{a}} + \mathcal{U}[8, 12], \text{ truncated to } [t^{\text{a}} + 8, 20] \quad \text{(departure time)}
\end{aligned}$$

where $\mathcal{N}(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2, and $\mathcal{U}[a, b]$ denotes the uniform distribution over the interval $[a, b]$.

HP parameters

HP parameters are generated based on real building data from the Zurich building database, including thermal conductance H (kW/°C), thermal capacitance C (kWh/°C), and maximum electricity power for heating $p^{\max}$ (kW). The ambient temperature profile q_t^{amb} is obtained from historical recorded weather data in Switzerland on 1 January 2021. The thermal retention coefficient α is calculated as $\alpha = e^{-\Delta_t/(RC)}$ where $R = 1/H$ is the thermal resistance. The following parameters are used for each

HP and building:

$$\begin{aligned}
\eta &= 3.85 \quad \text{(coefficient of performance)} \\
q^{\min} &= 17.0^\circ\text{C} \quad \text{(minimum temperature)} \\
q^{\max} &= 23.0^\circ\text{C} \quad \text{(maximum temperature)} \\
q^{\text{set}} &= 20.0^\circ\text{C} \quad \text{(setpoint temperature)} \\
q^{\text{init}} &= 20.0^\circ\text{C} \quad \text{(initial temperature)}
\end{aligned}$$

BSS parameters

Each BSS has the following characteristics:

$$\begin{aligned}
p^{\max} &\in \{25, 50\} \text{ kW with probabilities } \{0.5, 0.5\} \quad \text{(maximum input/output power)} \\
B &\in \{100, 200\} \text{ kWh with probabilities } \{0.5, 0.5\} \quad \text{(battery capacity)} \\
\eta^{\text{chg}} &\sim \mathcal{U}[0.95, 0.98] \quad \text{(charging efficiency)} \\
\eta^{\text{dis}} &\sim \mathcal{U}[0.96, 0.98] \quad \text{(discharging efficiency)} \\
\text{SOC}_{\text{init}} &\sim \mathcal{U}[0.3, 0.7] \quad \text{(initial SOC)} \\
e^{\text{init}} &= \text{SOC}^{\text{init}} \times B \quad \text{(initial energy)} \\
e^{\max} &= B \quad \text{(maximum energy)} \\
e^{\min} &= 0 \quad \text{(minimum energy)}
\end{aligned}$$

3.4 PolyFormer configuration and training settings

PolyFormer is used to approximate the aggregate power feasible region of the entire resource fleet over the 24-hour horizon. To ensure numerical stability during training, the aggregated power values are normalized to the interval $[0, 1]$ based on the total adjustable power range of all resources at each time period. Therefore, the error values reported in Fig. 2 of the main text are also normalized errors without units.

The $\mathbf{A}$ matrix defining the approximate polytope is an $M \times n$ matrix, where $n = T = 24$ is the number of time intervals, and M is set to $4T = 96$. The initial $\mathbf{A}$ matrix combines the power and accumulated energy constraints, which is defined as follows:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} \mathbf{I}_T \\ -\mathbf{I}_T \\ \mathbf{L}_T \\ -\mathbf{L}_T \end{bmatrix} \tag{S.13}$$

where $\mathbf{I}_T$ is the $T \times T$ identity matrix (instantaneous power constraints), and $\mathbf{L}_T$ is the $T \times T$ lower triangular matrix normalized by the square root of row sums to represent cumulative energy constraints:

$$L_{T,ij} = \begin{cases} \frac{1}{\sqrt{i}} & \text{if } 1 \le j \le i \le T \\ 0 & \text{otherwise} \end{cases} \tag{S.14}$$

We train PolyFormer under a constant learning rate, which is 2×10^{-2} in Scenario 1 and 1×10^{-2} in Scenario 2. The training employs a three-phase adaptive weighting strategy to control the feasibility and optimality objectives, as detailed in Table 6.

Table 6 Three-phase training strategy for resource aggregation

Phase	Epochs	`rate_opt_feas`	Focus
Phase 1: Balancing	1-500	1	Balanced feasibility and optimality
Phase 2: Refining	501-700	0.1	Emphasize feasibility
Phase 3: Converging	701-800	1×10^{-4}	Drive to internal approximation

Since there is no varying parameter in this case, we only use the `PreTrainNet` architecture without any input or hidden layer. Other training hyperparameters are summarized in the table below:

Table 7 Training hyperparameters for resource aggregation

Hyperparameter	Value
`optimizer`	SGD
`solver`	`gurobi`
`cvx_solver`	`gurobi`
`batch_size`	1
`n_cal`	3
`cal_feas`	True
`cal_opt`	True
`scheduler`	`StepLR` (step size=200, $\gamma = 0.98$)

4 Numerical settings for the two-layer power system optimization

4.1 Transmission network modelling

The optimization problem seeks to minimize the total generation cost of the transmission network while satisfying all operational constraints at both layers. The upper-layer transmission network is modelled using the standard AC power flow formulation, as follows:

$$\min \quad \sum\nolimits_{i \in \mathcal{N}_{\text{gen}}} a_i^2 P_i^g + b_i P_i^g \tag{S.15a}$$

$$\text{s.t.} \quad P_i^g - P_i^d = V_i \sum\nolimits_{j \in \mathcal{N}} V_j (G_{ij} \cos \phi_{ij} + B_{ij} \sin \phi_{ij}), \quad \forall i \in \mathcal{N}, \tag{S.15b}$$

$$Q_i^g - Q_i^d = V_i \sum\nolimits_{j\in\mathcal{N}} V_j(G_{ij}\sin\phi_{ij} - B_{ij}\cos\phi_{ij}), \quad \forall i \in \mathcal{N}, \tag{S.15c}$$

$$V_i^{\min} \le V_i \le V_i^{\max}, \quad \forall i \in \mathcal{N}, \tag{S.15d}$$

$$V_0 = V_{\text{ref}}, \tag{S.15e}$$

$$P_i^g = Q_i^g = 0, \quad \forall i \in \mathcal{N} \setminus \mathcal{N}_{\text{gen}}, \tag{S.15f}$$

$$P_i^{g,\min} \le P_i^g \le P_i^{g,\max}, \quad \forall i \in \mathcal{N}_{\text{gen}}, \tag{S.15g}$$

$$Q_i^{g,\min} \le Q_i^g \le Q_i^{g,\max}, \quad \forall i \in \mathcal{N}_{\text{gen}}, \tag{S.15h}$$

$$P_i^d = P_i^{d,0}, Q_i^d = Q_i^{d,0}, \quad \forall i \in \mathcal{N} \setminus \mathcal{N}_{\text{dist}}, \tag{S.15i}$$

$$(P_i^d, Q_i^d) \in \Omega_i(V_i), \quad \forall i \in \mathcal{N}_{\text{dist}}, \tag{S.15j}$$

where P_i^g and Q_i^g are the active and reactive power generation at node i; a_i and b_i are the cost coefficients for generator i; P_i^d and Q_i^d are the active and reactive power demand at node i; V_i is the voltage magnitude at node i; ϕ_{ij} is the voltage angle difference between nodes i and j; G_{ij} and B_{ij} are the conductance and susceptance of the line connecting nodes i and j; $\mathcal{N}$ is the set of all nodes; $\mathcal{N}_{\text{gen}}$ is the set of generator nodes; $\mathcal{N}_{\text{dist}}$ is the set of nodes connected to a flexible distribution network; V_{ref} is the reference voltage magnitude; $P_i^{g,\min}$, $P_i^{g,\max}$, $Q_i^{g,\min}$, and $Q_i^{g,\max}$ are the generation limits; $V_i^{\min}$ and $V_i^{\max}$ are the nodal voltage limits; $P_i^{d,0}$ and $Q_i^{d,0}$ are the fixed active and reactive demands at non-flexible nodes; and $\Omega_i(V_i)$ represents the feasible region of power injections to the distribution network connected at transmission node i, which depends on the voltage magnitude V_i. This corresponds to $\Omega(V)$ defined in the Method of the main text, except that the node index subscript i was omitted there for brevity. The set $\Omega_i(V_i)$ is specified by the operational constraints of the distribution network, and its explicit mathematical formulation is provided in the following subsection.

The objective function (S.15a) minimizes the total generation cost. The constraints (S.15b) and (S.15c) represent the AC power flow equations for active and reactive power balance at each node. The voltage limits are enforced by (S.15d), and the reference node voltage is set by (S.15e). Constraints (S.15f) ensure that non-generator nodes have zero generation. Generation limits are imposed by (S.15g) and (S.15h). Fixed demands at non-distribution nodes are specified in (S.15i), while the flexible demands at distribution-connected nodes are constrained by the feasible region $\Omega_i(V_i)$ in (S.15j).

4.2 Distribution network modelling

We here provide detailed formulations of the distribution network models to specify the feasible region of distribution networks $\Omega_i(V_i)$.

The distribution network models used in this study consist of eight balanced (single-phase) IEEE standard test networks and one unbalanced (three-phase) real-world network from Zhejiang Province, China. All networks are modelled using the DistFlow equations, with formulation differences arising from their balanced or unbalanced structures. The corresponding formulations are presented below.

Balanced distribution network (single-phase) formulation

$$\textbf{Variables:} \quad p_{ij}, q_{ij}, \ell_{ij} \in \mathbb{R}, \forall (i,j) \in \mathcal{E}, \quad v_j, p_i, q_i \in \mathbb{R}, \forall i \in \mathcal{N}$$

$$\sum_{k:j\to k} p_{jk} = p_{ij} - r_{ij} l_{ij} + p_j, \quad \forall (i,j) \in \mathcal{E} \tag{S.16a}$$

$$\sum_{k:j\to k} q_{jk} = q_{ij} - x_{ij} l_{ij} + q_j, \quad \forall (i,j) \in \mathcal{E} \tag{S.16b}$$

$$v_j = v_i - 2(r_{ij} p_{ij} + x_{ij} q_{ij}) + (r_{ij}^2 + x_{ij}^2) l_{ij}, \quad \forall (i,j) \in \mathcal{E} \tag{S.16c}$$

$$\ell_{ij} v_i = p_{ij}^2 + q_{ij}^2, \quad \forall (i,j) \in \mathcal{E} \tag{S.16d}$$

$$p_i^{\text{base}} - \delta_i |p_i^{\text{base}}| \le p_i \le p_i^{\text{base}} + \delta_i |p_i^{\text{base}}|, \quad \forall i \in \mathcal{N}_F \tag{S.16e}$$

$$q_i^{\text{base}} - \delta_i |q_i^{\text{base}}| \le q_i \le q_i^{\text{base}} + \delta_i |q_i^{\text{base}}|, \quad \forall i \in \mathcal{N}_F \tag{S.16f}$$

$$\underline{v} \le v_i \le \overline{v}, \quad \forall i \in \mathcal{N} \tag{S.16g}$$

$$v_0 = V_{\text{trans}}^2, \tag{S.16h}$$

where $\mathcal{N}$ and $\mathcal{E}$ denote the sets of nodes and distribution lines, respectively; 0 represents the slack node; $\mathcal{N}_F \subseteq \mathcal{N}$ denotes the set of nodes with flexible loads; the variables p_{ij} and q_{ij} denote the active and reactive power flow on line $(i,j) \in \mathcal{E}$, respectively; r_{ij} and x_{ij} denote the resistance and reactance of line (i,j); ℓ_{ij} denotes the squared current magnitude on line (i,j); v_i denotes the squared voltage magnitude at node i; p_j and q_j denote the active and reactive power injections at node j; p_i^{base} and q_i^{base} denote the base active and reactive power injections at node i; δ_i denotes the flexibility rate at node i; $\underline{v}$ and $\overline{v}$ denote the lower and upper voltage limits, respectively; V_{trans} denotes the voltage magnitude at the transmission-distribution (T-D) interface (the transmission node number is replaced by the subscript "trans" to avoid confusion with distribution node indices).

Unbalanced distribution network (three-phase) formulation

$$\textbf{Variables:} \quad p_{ij}^{\phi}, q_{ij}^{\phi}, \ell_{ij}^{\phi} \in \mathbb{R}, \forall (i,j) \in \mathcal{E}, \phi, \quad v_j^{\phi}, p_i^{\phi}, q_i^{\phi} \in \mathbb{R}, \forall i \in \mathcal{N}, \phi$$

$$\sum_{k:j\to k} p_{jk}^{\phi} = p_{ij}^{\phi} - \sum_{\psi} r_{ij}^{\phi\psi} \ell_{ij}^{\psi} + p_j^{\phi}, \quad \forall (i,j) \in \mathcal{E} \tag{S.17a}$$

$$\sum_{k:j\to k} q_{jk}^{\phi} = q_{ij}^{\phi} - \sum_{\psi} x_{ij}^{\phi\psi} \ell_{ij}^{\psi} + q_j^{\phi}, \quad \forall (i,j) \in \mathcal{E} \tag{S.17b}$$

$$v_j^{\phi} = v_i^{\phi} - 2 \sum_{\psi} \left(r_{ij}^{\phi\psi} p_{ij}^{\psi} + x_{ij}^{\phi\psi} q_{ij}^{\psi} \right) + \sum_{\psi} |z_{ij}^{\phi\psi}|^2 \ell_{ij}^{\psi}, \quad \forall (i,j), \phi \tag{S.17c}$$

$$(p_{ij}^{\phi})^2 + (q_{ij}^{\phi})^2 = \ell_{ij}^{\phi} v_i^{\phi}, \quad \forall (i,j) \in \mathcal{E}, \phi \tag{S.17d}$$

$$p_i^{\text{base},\phi} - \delta_i |p_i^{\text{base},\phi}| \le p_i^{\phi} \le p_i^{\text{base},\phi} + \delta_i |p_i^{\text{base},\phi}|, \quad \forall i \in \mathcal{N}_F, \phi \tag{S.17e}$$

$$q_i^{\text{base},\phi} - \delta_i |q_i^{\text{base},\phi}| \le q_i^{\phi} \le q_i^{\text{base},\phi} + \delta_i |q_i^{\text{base},\phi}|, \quad \forall i \in \mathcal{N}_F, \phi \tag{S.17f}$$

$$\underline{v} \leq v_i^{\phi} \leq \overline{v}, \quad \forall i \in \mathcal{N}, \phi \tag{S.17g}$$

$$v_0^{\phi} = V_{\text{trans}}^2, \quad \forall \phi \tag{S.17h}$$

where all variables and parameters are defined similarly to those in the balanced formulation, with the addition of phase superscripts $\phi, \psi \in \{a, b, c\}$ to denote the three phases; the impedance matrix structure is:

$$\mathbf{z}_{ij} = \begin{bmatrix} z_{ij}^{aa} & z_{ij}^{ab} & z_{ij}^{ac} \\ z_{ij}^{ba} & z_{ij}^{bb} & z_{ij}^{bc} \\ z_{ij}^{ca} & z_{ij}^{cb} & z_{ij}^{cc} \end{bmatrix} = \mathbf{r}_{ij} + \jmath \mathbf{x}_{ij} = \begin{bmatrix} r_{ij}^{aa} & r_{ij}^{ab} & r_{ij}^{ac} \\ r_{ij}^{ba} & r_{ij}^{bb} & r_{ij}^{bc} \\ r_{ij}^{ca} & r_{ij}^{cb} & r_{ij}^{cc} \end{bmatrix} + \jmath \begin{bmatrix} x_{ij}^{aa} & x_{ij}^{ab} & x_{ij}^{ac} \\ x_{ij}^{ba} & x_{ij}^{bb} & x_{ij}^{bc} \\ x_{ij}^{ca} & x_{ij}^{cb} & x_{ij}^{cc} \end{bmatrix}, \tag{S.18}$$

with the coefficient $\jmath$ of the reactance matrix $\mathbf{x}_{ij}$ being the imaginary unit.

The above constraints (single-phase or three-phase) jointly define the feasible region of the distribution network. However, from the perspective of the transmission network, only feasible active and reactive power region at the T-D interface, i.e., the root node of the distribution network, is of interest. This feasible region is induced by the active/reactive power ranges of the flexible nodes within the distribution network, while simultaneously constrained by distribution power flow constraints and influenced by voltage variations at the T-D interface V_{trans}. Accordingly, it is represented as $\Omega(V_{\text{trans}}) \subseteq \mathbb{R}^2$.

4.3 T-D networks configuration

We considered three transmission networks from the IEEE standard test cases. Each transmission network differs in node count, line count, and the number of attached distribution networks; the detailed configurations are provided in Table 8. For each transmission case we made modifications to generation capacities, base load levels, and line impedances so that the transmission networks can be coupled with the selected distribution networks while preserving the feasibility of the AC power-flow computations.

Table 8 Transmission network configurations

Network label	Name in the code	Number of nodes	Number of distribution networks
A	`case4gs_ts`	4	3
B	`case118_ts`	118	72
C	`case300_ts`	300	149

For the distribution side we included eight balanced (single-phase) IEEE distribution test networks and one real three-phase unbalanced distribution network from Zhejiang Province, China. Table 9 summarizes these distribution networks. In every distribution network, the top 50% of load nodes are defined as flexible, and their flexibility rates $\delta_i (\forall i \in \mathcal{N}_F)$ are all set to 0.3. Similar to the transmission models, load

injections and line impedances for each distribution network were modified to ensure power flow feasibility when coupled to the transmission cases.

Table 9 Distribution network configurations

Network label	Name in the code	Number of nodes	Phases
a	`case10ba_ds`	10	Single
b	`case17me_ds`	17	Single
c	`case33bw_ds`	33	Single
d	`case51ga_ds`	51	Single
e	`case74_ds`	74	Single
f	`case118zh_ds`	118	Single
g	`case136ma_ds`	136	Single
h	`case533mt_hi_ds`	533	Single
i	`case36real_3phase_ds`	36	Three

By connecting the distribution networks a-i to the T-D interface nodes of transmission networks A-C, we constructed $3 \times 9 = 27$ T-D test cases as shown in Fig. 3 of the main text.

4.4 PolyFormer configuration and training settings

PolyFormer is used to approximate $\Omega(V_{\text{trans}})$ (varying parameter V_{trans}) for each distribution network. The number of rows of the $\mathbf{A}$ matrix is set to $M = 8$, and the number of columns is $n = 2$ (active power p_0 and reactive power q_0). The initial $\mathbf{A}$ matrix is defined as follows:

$$\mathbf{A}_{\text{init}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \\ -1 & -1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{S.19}$$

The dataset for the varying parameter V_{trans} is constructed from 100 uniformly distributed sampling points over the interval [0.95, 1.05] p.u. For each distribution network, we first apply `PreTrainNet` to identify the polytope that best fits the shape of $\Omega(V_{\text{trans}})$, and subsequently use `FullNet` to establish the mapping between V_{trans} and the approximating polytope.

The learning rates for both pretraining and full training are set as case-adaptive values. For pretraining, the `lr` is defined as $0.1/p_0^{\text{base}\ 3}$, the weighting coefficient `rate_opt_feas` is set to 1, and the number of training iterations `n_train` is set to 500. The full network utilizes two different training strategies. One strategy outputs the "Moderate PolyFormer", where the `rate_opt_feas` is set to 0.6, the `lr` is set to $5 \times 10^{-5}/p_0^{\text{base}}$, and the number of training iterations is `n_train` $= 60$. The second strategy outputs the "Feasible PolyFormer", which include two phases with different

`rate_opt_feas` and `lr` to control the feasibility and optimality objectives, as detailed in Table 10:

Table 10 Two-phase training strategy of the `FullNet` for distribution network constraints

Phase	Epochs	`rate_opt_feas`	`lr`
Phase 1: Balancing	1-40	1	$5 \times 10^{-5}/p_0^{\text{base}}$ (cases a-c), $1 \times 10^{-5}/p_0^{\text{base}}$ (cases d-i)
Phase 2: Converging	41-50	1×10^{-3} (cases a-g, i), 1×10^{-2} (case h)	$5 \times 10^{-5}/p_0^{\text{base}}$ (cases a-c), $2 \times 10^{-6}/p_0^{\text{base}}$ (cases d-g, i), $1 \times 10^{-6}/p_0^{\text{base}}$ (case h)

All other hyperparameters are identical across cases, as summarized in Table 11.

Table 11 Training hyperparameters in the two-layer power system optimization

Model types	`PreTrainNet`	`FullNet`
`optimizer`	SGD	Adam
`solver`	`ipopt`	`ipopt`
`cvx_solver`	`gurobi`	`gurobi`
`n_hidden`	0	128
`batch_size`	1	1
`n_cal`	5	2
`cal_feas`	True	True
`cal_opt`	True	True
`scheduler`	StepLR (step size=100, gamma=0.98)	StepLR (step size=100, gamma=0.95)

4.5 Performance evaluation

We evaluate the performance of PolyFormer on the 27 test cases for the two-layer power system optimization following the steps below.

1 We apply PolyFormer to fit the feasible region $\Omega(V_{\text{trans}})$ of each type of distribution networks.
2 Assuming that all distribution network nodes operate at their baseline power, the distribution network power-flow equations are solved to obtain the total active and reactive power at the root node of each distribution network. These active and reactive power values are then assigned to the corresponding T-D interface nodes in the transmission network, after which the transmission-network power-flow equations

[3]Here, p_0^{base} denotes the base active power injection at the root node of the distribution network. This scaling is applied to ensure numerical stability during training across different distribution networks with varying power injection levels.

are solved to determine the initial voltage estimates at all T-D interfaces, denoted by $\widehat{V}_i$.

3 The initial voltage estimate $\widehat{V}_i$ is fed into the corresponding distribution-network PolyFormer to instantiate a polytope approximation of its active and reactive power feasible region.

4 The resulting coefficients $\mathbf{A}_i(\widehat{V}_i)$ and $\mathbf{b}_i(\widehat{V}_i)$ are held fixed throughout the subsequent transmission-network optimization and embedded as $\mathbf{A}_i(\widehat{V}_i)[P_i^d, Q_i^d]^\top \leq \mathbf{b}_i(\widehat{V}_i)$. This one-shot treatment approximates the voltage dependence of the distribution-network feasible regions. Solving the simplified problem yields the objective value f as well as the active and reactive powers at the root node of each distribution network, denoted by $(p_{i,0}^*, q_{i,0}^*)$ (the index i represents the node in the transmission network, and the index 0 represents the root node of the distribution network).

5 For every distribution network i in a T-D case, we take the obtained $(p_{i,0}^*, q_{i,0}^*)$ as inputs to solve Problem (S.20) for the feasibility error ErrFeas, with the interface voltage fixed at the initial estimate $\widehat{V}_i$ used to instantiate the polytope. After the computation for all distribution networks in the case completes, we record the maximum feasibility error.

$$\begin{aligned} \text{ErrFeas} = \min \quad & (p_0 - p_{i,0}^*)^2 + (q_0 - q_{i,0}^*)^2 \\ \text{s.t. for balanced networks: } & \text{(S.16a)} - \text{(S.16h)} \\ \text{for unbalanced networks: } & \text{(S.17a)} - \text{(S.17h)} \end{aligned} \tag{S.20}$$

As shown in Fig. 3c of the main text, this one-shot approximation yields small measured feasibility errors. For Feasible PolyFormer, the maximum feasibility error is below 10^{-8} in every test case.

6 For the objective function, we compare the relative change in the objective value obtained using PolyFormer against the value obtained by directly solving the full two-layer power system optimization problem with `IPOPT`. This yields the objective error metric.

7 Finally, we record the solver time and peak memory usage for the simplified optimization in Step 4. These measurements exclude the baseline power-flow calculations in Step 2 and neural-network inference in Step 3.

5 Numerical settings for DRCC portfolio optimization

5.1 Data-driven conditional value-at-risk reformulation of the DRCC portfolio optimization

In the DRCC portfolio optimization problem (as formulated in Equation (16) of the main text), the DRCC (16b) includes an infinite number of constraints due to the ambiguity set defined by the Wasserstein metric. For each asset group g, let $\mathcal{Q}_g$ denote the Wasserstein ambiguity set with radius ρ_g, centred at the empirical distribution of its K historical return samples. To facilitate the approximation, we first reformulate it into a tractable form based on the data-driven conditional value-at-risk (CVaR)

approximation proposed in [1]. The reformulated DRCC is as follows:

$$\begin{aligned}
&\upsilon_g \rho_g + \frac{1}{K}\sum_{k=1}^{K} s_{gk} \leq 0, \\
&\beta_g \leq s_{gk}, \forall k \leq K, \\
&-\mathbf{x}_g^\top \hat{\mathbf{r}}_{gk} + R_g \mathbf{1}^\top \mathbf{x}_g + (\epsilon_g - 1)\beta_g + \epsilon_g \boldsymbol{\gamma}_{gk}^\top (\mathbf{h} - \mathbf{H}\hat{\mathbf{r}}_{gk}) \leq \epsilon_g s_{gk}, \forall k \leq K, \\
&-\epsilon_g \upsilon_g \leq \epsilon_g \mathbf{H}^\top \boldsymbol{\gamma}_{gk} + \mathbf{x}_g \leq \epsilon_g \upsilon_g, \forall k \leq K
\end{aligned} \tag{S.21}$$

where $\mathbf{x}_g \in \mathbb{R}^{N_g}$ are the decision variables representing the investment proportions in asset group g, with $\mathcal{N}_g$ denoting the set of indices for the assets in group g, and N_g being the number of assets in group g; $\hat{\mathbf{r}}_{gk} \in \mathbb{R}^{N_g}$ denotes the vector of k-th ($\forall k \leq K$) historical sample for the uncertain returns $\tilde{\mathbf{r}}_g$ of the assets in group g; $\mathbf{s}_g \in \mathbb{R}^K$, $\upsilon_g \in \mathbb{R}$, $\beta_g \in \mathbb{R}$, and $\boldsymbol{\gamma}_{gk} \in \mathbb{R}_+^{\dim(\mathbf{h})}, \forall k \leq K$ are auxiliary variables introduced in the reformulation; ϵ_g, ρ_g, and R_g are varying parameters, which represent risk level, Wasserstein radius, and minimum return for asset group g, respectively, as explained in the "Apply PolyFormer to DRCC portfolio optimization" section in the Methods of the main text; the matrix $\mathbf{H} \in \mathbb{R}^{2N_g \times N_g}$ and vector $\mathbf{h} \in \mathbb{R}^{2N_g}$ define the support set of the uncertain asset returns, which is a hyperrectangle:

$$\{\tilde{\mathbf{r}}_g \in \mathbb{R}^{N_g} | \mathbf{H}\mathbf{r}_g \leq \mathbf{h}\}, \quad \mathbf{H} = \begin{bmatrix} \mathbf{I}_{N_g} \\ -\mathbf{I}_{N_g} \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} 0.1 \cdot \mathbf{1}_{N_g} \\ 0.1 \cdot \mathbf{1}_{N_g} \end{bmatrix} \tag{S.22}$$

where $\mathbf{I}_{N_g}$ is the identity matrix of size N_g, and $\mathbf{1}_{N_g}$ is a vector of ones of size N_g. According to this definition, all the uncertain asset returns $\tilde{\mathbf{r}}_g$ are bounded within the range $[-10\%, +10\%]$, and $\dim(\mathbf{h}) = 2N_g$.

By applying the reformulated DRCC constraints (S.21) for all asset groups $g \in \mathcal{G}$, we obtain a set of linear inequalities that can be used as a surrogate for the original DRCC constraints (16b) in the portfolio optimization problem, which is the "DRCC-linear" model referred to in the main text.

Although the reformulated DRCC constraints (S.21) are linear, they still involve a large number of auxiliary variables and constraints, especially when the number of historical samples K is large. This results in high computational burdens when repeatedly solving the DRCC-linear model for varying parameter values to obtain multiple portfolio strategies with different risk-return profiles. To address this issue, PolyFormer is employed to approximate the feasible region of decision variables $\mathbf{x}_g$ for each asset group g, which is defined not only by the reformulated DRCC constraints (S.21) but also by the group investment cap constraint (16c) in the main text. Hence, the varying parameters for each asset group g are $\boldsymbol{\theta}_g = (\epsilon_g, \rho_g, R_g, X_g^{\max})$.

5.2 DRCC configuration

In the four numerical experiments of the DRCC portfolio optimization problem presented in Fig. 4 of the main text, the number of assets N, asset groups G, and historical samples K are set as follows:

- **Case a**: $N = 50$ assets, $G = 2$ groups, $K = 150$ historical samples;
- **Case b**: $N = 150$ assets, $G = 3$ groups, $K = 300$ historical samples;
- **Case c**: $N = 300$ assets, $G = 5$ groups, $K = 900$ historical samples;
- **Case d**: $N = 400$ assets, $G = 8$ groups, $K = 1280$ historical samples.

Each asset group g contains an equal number of assets, i.e., $N_g = \frac{N}{G}, \forall 1 \leq g \leq G$.

The uncertain asset returns are generated using truncated normal distributions with the following characteristics:

- **Return distributions**: Each group g has distinct return characteristics:

$$\begin{aligned}
\mu_g &= 0.03 + 0.01\frac{g-1}{G-1}, 1 \leq g \leq G \quad \text{(expected return)} \\
\sigma_g &= 0.01 + 0.02\frac{g-1}{G-1}, 1 \leq g \leq G \quad \text{(standard deviation)} \\
\tilde{r}_i &\sim \text{TruncatedNormal}(\mu_g, \sigma_g, -0.1, 0.1) \quad \forall i \in \mathcal{N}_g
\end{aligned}$$

- **Historical data for the uncertain returns**: K historical samples are generated for each asset using truncated normal distributions with bounds $[-0.1, 0.1]$.
- **Test data for the uncertain returns**: Additional $K_{\text{test}} = 100$ samples are generated for out-of-sample performance evaluation.

Each asset group g has four varying parameters $\boldsymbol{\theta}_g = (\epsilon_g, \rho_g, R_g, X_g^{\max})$. Each of these parameters varies within specific ranges, so the dataset for these parameters is generated from uniform distributions on these ranges, and the size of the dataset is set to 100 per asset group. The initial value and varying ranges for each parameter are listed below:

Table 12 Varying parameters for DRCC portfolio optimization

Parameter	Symbol	Initial Value	Varying Range
Risk level	ϵ_g	0.10	[0.07, 0.13]
Wasserstein radius	ρ_g	10^{-4}	$[10^{-6}, 10^{-3.5}]$
Minimum return	R_g	Adaptive[1]	Adaptive range[1]
Group investment cap	$X_g^{\max}$	Adaptive[2]	Adaptive range[2]

5.3 PolyFormer configuration and training settings

The $\mathbf{A}$ matrix defining the approximate polytope for each asset group g is an $M \times N_g$ matrix, where N_g is the number of assets in group g, and M is set to $4N_g + 2$, which

[1] R_g is initialized as $\max(-0.1, \mu_g - 1.5\sigma_g)$, where μ_g and σ_g are the mean and standard deviation of asset returns in group g. The varying range is $[\max(-0.1, \mu_g - 1.6\sigma_g), \max(-0.1, \mu_g - 1.4\sigma_g)]$.

[2] $X_g^{\max}$ is initialized as the midpoint of $[\min(1.0, 1.0/G + 0.2), \min(1.0, 1.0/G + 0.4)]$, with this interval serving as the varying range.

is initialized as follows:

$$\mathbf{A}_{\text{init}} = \left[\mathbf{I}_n,\ -\mathbf{I}_n,\ \frac{\mathbf{1}\mathbf{1}^\top - \mathbf{I}_n}{\sqrt{n-1}},\ -\frac{\mathbf{1}\mathbf{1}^\top + \mathbf{I}_n}{\sqrt{n-1}},\ \frac{\mathbf{1}}{\sqrt{n}},\ -\frac{\mathbf{1}}{\sqrt{n}}\right]^\top \tag{S.23}$$

where $\mathbf{1}$ is a column vector of ones of size n, so $\mathbf{1}\mathbf{1}^\top$ is a matrix of ones of size $n \times n$.

In the DRCC portfolio-optimization case, the DRCC itself is designed to handle uncertainty, and the uncertainty distribution inferred from historical data naturally deviates from the distribution of the test data. As a result, the distinction between internal and external approximations becomes less critical. Therefore, we directly use the initialized outer polytope as the output of the `PreTrainNet` without performing any additional pretraining. In other words, we train the model using only the `FullNet` architecture in this case. The training hyperparameters are listed in the table below:

Table 13 Training hyperparameters for DRCC portfolio optimization

Hyperparameter	Value
`n_train`	30
`optimizer`	Adam
`lr`	2×10^{-4}
`solver`	`gurobi`
`cvx_solver`	`gurobi`
`n_hidden`	128
`batch_size`	5
`n_cal`	2
`cal_feas`	True
`cal_opt`	True
`rate_opt_feas`	1
`scheduler`	StepLR (step size=100, gamma=0.97)

5.4 Performance evaluation

After training, the performance of PolyFormer is evaluated on a test dataset consisting of $N_{\text{test}} = 100$ samples of the uncertain return $\tilde{\mathbf{r}}_g$, representing realized asset returns, as described in Supplementary Subsection 5.2. The evaluation procedure is as follows:

1 Randomly generate a parameter configuration within the varying ranges of ϵ_g, ρ_g, R_g, and $X_g^{\max}$ for each asset group.
2 Feed the parameter $\boldsymbol{\theta}_g = (\epsilon_g, \rho_g, R_g, X_g^{\max})$ into the trained $\mathbf{A}$-net and $\mathbf{b}$-net models to obtain the approximated polytope feasible region for the decision variable $\mathbf{x}_g$ of each asset group g.
3 Replace constraints (16b) and (16c) in the portfolio optimization model of the main text with the polytopic approximations generated by PolyFormer, and solve the resulting optimization problem to obtain the investment portions $\mathbf{x}_g$ for each asset group.

4 Evaluate the resulting investment portions on the test dataset by computing the average violation of the minimum-return constraint and the average realized return.
5 Repeat Steps 1-4 for $n_{\text{test}} = 300$ independent trials, thereby obtaining the distributions of constraint violation values and realized returns under varying parameter configurations.

References

[1] Ordoudis, C., Nguyen, V.A., Kuhn, D., Pinson, P.: Energy and reserve dispatch with distributionally robust joint chance constraints. Operations Research Letters **49**(3), 291–299 (2021)