Supplementary Material

Supplementary Information for
A Patch-Routed Mixture-of-Experts for Continual MOBA Draft Recommendation

This document supports the submitted manuscript: the environment and configuration in full, the per-seed values behind every table, recovery behaviour for all seeds rather than one, the zero-forgetting verification, the measured computational cost, and the code central to the method.

Contents

1 S0: Reproducibility metadata

1.1 Software environment

Source: `environmentt.yml` (conda env `cl`), verified 2026-08-01.

Component	Version
OS	Windows 10, build 10.0.19045 (21H2/22H2)
Python	3.11.13
PyTorch	2.5.1+cu121
CUDA (torch build)	12.1
torchvision / torchaudio	0.20.1+cu121 / 2.5.1+cu121
Avalanche	avalanche-lib 0.6.0
RecBole	1.2.1
NumPy	2.4.4
torchmetrics	1.8.2

Compute backend: CUDA.

1.2 Hardware

Runs were distributed across **three machines**. Matched pairs (Base and PatchExpertFFN at the same seed) ran on the same machine, so the within-pair runtime comparison in S5 is valid; absolute times are not comparable across seeds.

Assignment is exact, taken from the per-run metadata recorded by the experiment tracker, not inferred.

Seed	GPU	VRAM	Cores	RAM	Driver CUDA
1	NVIDIA GeForce GTX 1660 SUPER (Turing)	6 GB	10	31 GB	13.3
2	NVIDIA GeForce GTX 1660 (Turing)	6 GB	10	31 GB	12.5
3	NVIDIA GeForce RTX 2060 SUPER (Turing)	8 GB	16	63 GB	13.3

All three run Windows 10 build 19045 and the same conda environment. The two arms of a given seed always ran on the same machine, which is what makes the within-seed runtime comparison in S5 meaningful; the three machines differ, so absolute times are not comparable across seeds.

1.3 Seed policy

Seeds 1, 2, 3, reported as mean ± sample standard deviation (n = 3). `cl_localize/utils.py::set_all_seeds` sets, per run:

```
random.seed(seed); np.random.seed(seed)
torch.manual_seed(seed); torch.cuda.manual_seed_all(seed)
```

So the seed governs weight initialization, batch ordering, and BERT mask sampling.

The train/valid/test split is not seeded by the run seed. It uses a fixed `random.seed(42)` (`cl_localize/utils.py:138,146`), so all three seeds share a byte-identical split. Cross-seed variance therefore isolates training stochasticity from data-split variation.

Runs are not bit-deterministic. The driver sets `torch.backends.cudnn.benchmark = True` (line 1242) and does not set `cudnn.deterministic = True`. cuDNN autotuning selects kernels at runtime, so re-running an identical seed can yield small numerical differences.

This does **not** affect the zero-forgetting result in S4. That claim is bit-identity between two checkpoints written by the *same* process (expert j saved at experience j versus at experience 4), with no retraining in between, so nondeterminism cannot affect it.

1.4 Checkpoint naming

`checkpoints/per_exp/<arm>_<seed>_exp<N>.pt`, for example `Base_optprobe_seed1_exp1.pt`. Files are per-experience snapshots used by `eval_bwt_posthoc.py` for the S4 verification.

1.5 W&B field names used

The paper's numbers come from specific run-summary fields. Named here because two similarly titled fields exist and only one is correct.

Paper quantity	Field
Plateau HR@10 (Table 3)	`recovery/final_hit@10_exp{1..4}`
Recovery samples (Table 3)	`recovery/steps_exp{1..4}`
Wall-clock (S5)	`_runtime`

2 S1: Experimental configuration

Source: `cl_localize/config.py` and `cl-localize perturbation.py`, verified against the model as instantiated (see S0). Extends Appendix A rather than repeating it.

2.1 Model

Parameter	Value	Config symbol
Backbone	BERT4Rec, opponent-aware	`MODEL_TYPE`
Hidden dimension	128	`D_MODEL`
Encoder layers	5	`N_LAYERS`
Attention heads	8	`N_HEAD`
Feed-forward width	512 (4d)	,
Dropout	0.2	`DROPOUT`
Sequence length	24 (full Captain's Mode draft)	`SEQ_LEN`
Hero catalogue	126	`NUM_ITEMS`
Mask token index	127	`MASK_TOKEN_ID`
Embedding rows	128 (padding 0, heroes 1–126, mask 127)	,
Normalization	pre-norm (`norm_first=True`)	,
Activation	GELU	,

Encoder depth was selected by comparison over $\{3, 5, 8\}$ layers; 5 was retained.

Parameter accounting, verified by instantiation:

Quantity	Value
Single-patch model `N_model`	1,077,247
Frozen trunk	401,024 (of which 3,584 never trainable)
Expert + head per patch	676,223
Storage fraction ρ	0.628
Full 5-patch library, end of stream	3,782,139

The three fixed context embeddings (team 256, action 256, draft order 3,072) are initialized analytically and have `requires_grad = False` in every arm.

2.1.1 Checkpoint size against the storage accounting

The reference implementation **pre-allocates the entire expert library at construction**: `num_patches` feed-forward experts in every encoder layer and `num_patches` output heads, sized from a fixed patch count. Nothing is added as experiences arrive.

A checkpoint file therefore holds all 3,782,139 parameters at every experience, not the subset trained so far. On disk:

	Params	fp32 size
Base, any experience	1,077,247	4.11 MB
PatchExpertFFN, any experience	3,782,139	14.43 MB

Table 5 reports something different, and deliberately: the storage a deployment must retain after training through patch T, which is `trunk + (T+1) × (expert + head)` = 401,024 + (T+1) × 676,223. The two agree only at $T = 4$.

T	Table 5	Checkpoint on disk
0	4.11 MB	14.43 MB
1	6.69 MB	14.43 MB
2	9.27 MB	14.43 MB
3	11.85 MB	14.43 MB
4	14.43 MB	14.43 MB

Eager allocation is an implementation convenience, not a property of the method. A per-patch expert is meaningless before its patch exists, and a deployment allocates one when the patch ships. The accounting in Table 5 is the deployable cost; the file size is an artifact of allocating the library up front so that routing indices are valid from the first forward pass.

The distinction matters when reading the deposit: comparing `ls` output against Table 5 will appear to contradict it at $T < 4$, and does not.

2.2 Optimization

Parameter	Value
Optimizer	Adam
Learning rate	0.001 (`PLASTICITY_LR`)
Train batch size	256 (`train_batch_size`)
Eval batch size	1024 (`eval_batch_size`)
Device	CUDA

Optimizer state is reset at every patch boundary for the routed and shared-weight arms. The sequential fine-tuning anchor carries its Adam moment estimates across boundaries.

2.3 Data

Item	Value
Dataset	DOTA-Draft, RecBole format
Fraction used	1.0, the full dataset (`DATASETSIZE`)
Split	train / valid / test
Split RNG	**fixed `random.seed(42)`**, independent of the run seed

Because the split seed is fixed, all three run seeds see a byte-identical split. See S0 for why this matters.

Patch 0 is trained once and shared across seeds. Its checkpoint is a single file (`checkpoints/base_exp0__PatchExpertFFNBert4Rec.pt`), not one per seed; `checkpoints/per_exp/` holds only experiences 1 to 4 for each seed. Seeding therefore applies to patches 1 to 4, and Exp0 has zero cross-seed variance by construction. This is consistent with Table 3, which excludes Exp0 as initial convergence rather than a drift event, and it is why HR_diag[0] in S4 is 0.6854 for all three seeds.

2.4 Patch stream

Five experiences, domain-incremental, in release order. Each experience is split 80/10/10 train/validation/test.

Experience	Patch	Dota 2 version	Matches	Interactions
Exp000	54	7.35	11,920	286,080
Exp001	55	7.36	6,057	145,368
Exp002	56	7.37	10,707	256,968
Exp003	57	7.38	6,912	165,888
Exp004	58	7.39	17,309	415,416
Total			52,905	1,269,720

Draft structure is fixed by the game and encoded as three constant sequences of length 24 (`DRAFT_TEAM_SEQ`, `DRAFT_ACTION_SEQ`, `DRAFT_ORDER_SEQ`): two teams alternating bans and picks, twelve actions each.

2.5 Arms

Arm	Run-name stem
Sequential fine-tuning (Base)	`Base`
PatchExpertFFN + per-patch head	`PatchExpertFFN_ppHead`
Model zoo	not run; identical by construction to Base's diagonal
Shared-weight reference	`FreezeExceptAttn`, `FreezeExceptFFN`

2.6 Framework

Avalanche 0.6.0 supplies the domain-incremental benchmark and the plugin system; PatchExpertFFN is implemented as an Avalanche plugin (`cl_localize/plugins/patch_expert_ffn.py`) over the model in `cl_localize/models/patch_expert_ffn_bert4rec.py`.

3 S2: Numerical results, per seed

Every value behind a mean ± standard deviation in Table 3 of the manuscript. The bold rows reproduce Table 3 exactly, to four decimal places for HR@10 and to the unit for recovery samples.

Statistics are the sample mean and sample standard deviation over n = 3 seeds. **No significance tests are reported.** With three seeds a t-test or Wilcoxon has no power, and the manuscript's stated protocol (§5.5) is non-overlap of seed ranges, not a p-value. Per-seed values are given so the reader can see the spread directly.

3.1 Plateau HR@10

W&B field `recovery/final_hit@10_exp{1..4}`.

Base (sequential fine-tuning)

Seed	Exp1	Exp2	Exp3	Exp4
1	0.6636	0.7065	0.7036	0.7172
2	0.6662	0.7083	0.7089	0.7191
3	0.6697	0.7118	0.7060	0.7155
mean ± sd	**0.6665 ± 0.0031**	**0.7088 ± 0.0027**	**0.7062 ± 0.0026**	**0.7173 ± 0.0018**

PatchExpertFFN+head

Seed	Exp1	Exp2	Exp3	Exp4
1	0.6698	0.7035	0.6935	0.6983
2	0.6676	0.6988	0.6917	0.6978
3	0.6713	0.6970	0.6962	0.6993
mean ± sd	**0.6695 ± 0.0019**	**0.6997 ± 0.0034**	**0.6938 ± 0.0023**	**0.6985 ± 0.0008**

At Exp1 the two arms' seed ranges overlap (0.6636 to 0.6697 against 0.6676 to 0.6713), which is the basis for the manuscript's statement that the plateaus are indistinguishable there. From Exp2 onward the ranges are disjoint and PatchExpertFFN+head sits below.

3.2 Recovery samples

W&B field `recovery/steps_exp{1..4}`: post-patch training samples to reach 95% of the arm's own plateau.

Base (sequential fine-tuning)

Seed	Exp1	Exp2	Exp3	Exp4
1	177,208	228,344	111,616	463,912
2	199,736	205,816	128,000	324,608
3	193,592	224,248	117,760	310,272
mean ± sd	**190,179 ± 11,645**	**219,469 ± 12,000**	**119,125 ± 8,277**	**366,264 ± 84,869**

PatchExpertFFN+head

Seed	Exp1	Exp2	Exp3	Exp4
1	93,184	195,584	152,664	353,320
2	84,992	168,960	154,712	381,992
3	99,328	177,152	158,808	341,032
mean ± sd	**92,501 ± 7,192**	**180,565 ± 13,636**	**155,395 ± 3,128**	**358,781 ± 21,019**

Exp1 seed ranges are disjoint by a wide margin (Base 177,208 to 199,736 against 84,992 to 99,328), which is the strongest single result in the table. Exp4's Base spread is large (±84,869, driven by seed 1 at 463,912), so the Exp4 comparison is correctly described in the manuscript as indistinguishable rather than favourable.

3.3 Backward transfer

Not tabulated here. BWT is computed post-hoc from saved checkpoints rather than from the training-time logs; see S4.

4 S3: Recovery behaviour, all seeds

Figure 6 in the manuscript shows seed 1 only. This section gives the same quantity for every seed and every patch transition.

4.1 What is plotted

x = training samples since the patch boundary (`recovery/since_drift_exp{N}`), y = held-out HR@10 on the just-trained patch (`recovery/hit@10_exp{N}`). These are the pair the recovery tracker in `cl_localize/metrics/recoverytime.py` records, and the same pair `plot_recovery_curves` uses for Figure 6.

39,116 points across 24 series (2 arms × 3 seeds × 4 transitions). Endpoints were checked against each run's summary and match exactly.

Presentation smoothing is a centred moving average of width 9 over a densely sampled curve. This is cosmetic only; the recovery statistic in Table 3 uses w = 3 on the tracker's own grid and is unchanged.

4.2 Figures

File	Use
`figures/recovery_curves_all_seeds_logx.{pdf,png}`	**primary.** Log x-axis
`figures/recovery_curves_all_seeds.{pdf,png}`	linear x, matches Figure 6's axis

The log axis is the one to publish. All of the recovery happens below 500,000 samples on axes that run to between 1.5 and 7 million, so on a linear scale the entire speed comparison is compressed into the leftmost few percent of the panel and is not readable. The linear version is kept only because it shares Figure 6's axis convention.

4.3 Speed, measured from the curves

Samples needed to first reach 95% of the arm's own final value. Computed directly from the raw curves, independent of Table 3.

Transition	seed 1	seed 2	seed 3	Manuscript claim
Exp1	0.43×	0.43×	0.57×	"roughly half as many post-patch samples"
Exp2	0.79×	0.93×	0.81×	"speed advantage persists at Exp2"
Exp3	1.42×	1.27×	1.67×	"reverses at Exp3"
Exp4	0.85×	1.07×	1.44×	"indistinguishable at Exp4"

Ratios below 1 favour PatchExpertFFN+head. Every claim in §6.1 holds on every seed, and the Exp1 figure lands on "roughly half" without rounding help.

4.4 Two things a reader will notice, and the answers

The curves end at different x. Early stopping triggers at different points per arm and seed, so a series stops where its run stopped. This is not truncation for display. Example: PatchExpertFFN+head seed 2 Exp1 ends at 1,045,952 samples while Base seed 3 Exp1 runs to 2,207,728.

PatchExpertFFN starts above Base at Exp2 and Exp3. Visible at the left of the log-scale panels. This is the copy-chain warm start: expert t is initialized from expert $t{-}1$, so it begins nearer the new patch than a freshly perturbed shared model does. It is a property of the design, not a measurement artifact.

4.5 Draft caption

> **Recovery curves for all three seeds.** Held-out HR@10 on the just-trained > patch against training samples since the patch boundary, log scale. Rows are > seeds, columns are patch transitions. PatchExpertFFN+head reaches its own > plateau from roughly half the post-patch samples at Exp1 on every seed, holds > a smaller advantage at Exp2, and is slower from Exp3 onward. Curves terminate > where early stopping halted the corresponding run. Manuscript Figure 6 is the > seed 1 row.

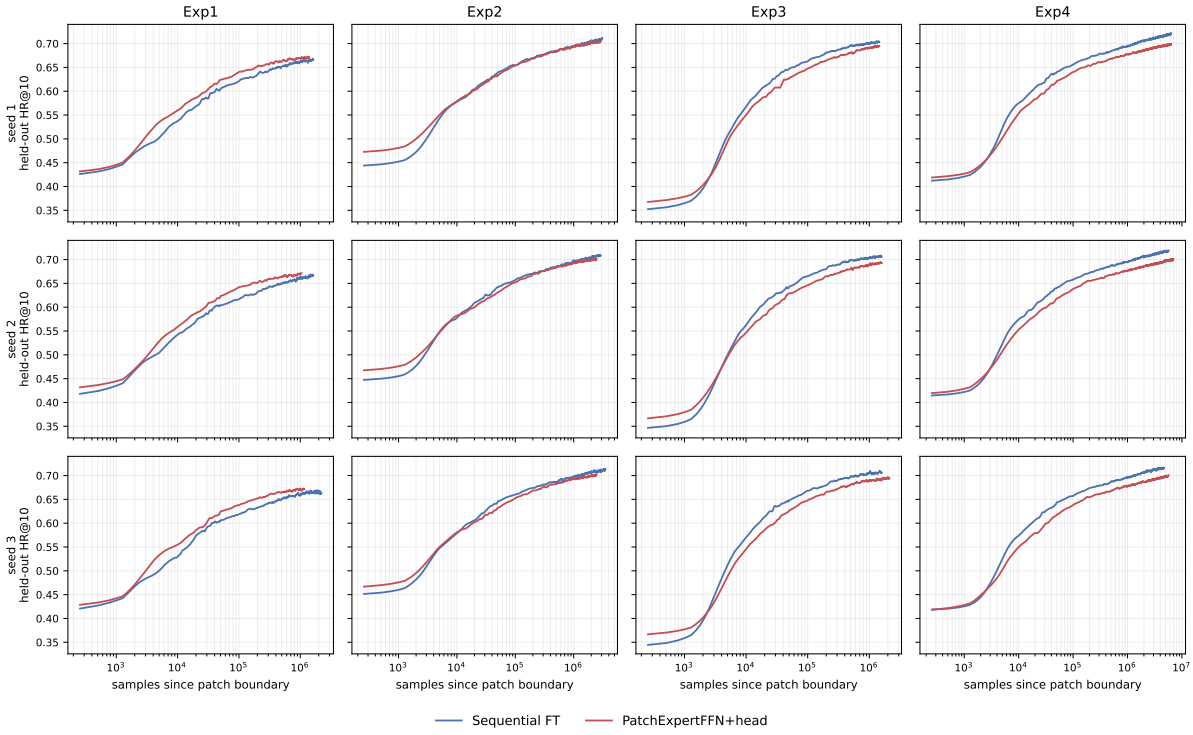

Figure 1: Recovery curves for all three seeds, log scale.

5 S4: Zero-forgetting verification

Source: `eval_bwt_posthoc.py`, run over `checkpoints/per_exp/`. Recomputed from saved checkpoints with **no retraining**, so it can be repeated at any time by anyone holding the deposit.

The manuscript claims backward transfer is *exactly* zero under routing, not approximately zero. This section is the evidence.

5.1 Method

For each seed and each past patch j:

1. **HR_diag[j]** is HR@10 on patch j's held-out split, measured with the model as it stood immediately after training on patch j. 2. **HR_final_routed[j]** is HR@10 on the same split at the end of the stream, scoring patch j through *its own* expert and head. 3. **HR_unrouted[j]** is the same, but scoring every past patch through the final (patch-4) expert and head. This is the ablation, not the method. 4. **BWT** is the mean of the per-patch deltas over $j = 0..3$.

Both terms use the same dataset instance. `NewHeroActionBERTDataset` enumerates mask positions deterministically with no RNG at construction, so the evaluation sets are byte-identical between the two measurements.

5.2 Results

Per-seed tables, HR@10.

5.2.1 Seed 1

Patch j	HR_diag	HR_final_routed	routed delta	bit-identical	HR_unrouted	unrouted delta
0	0.6854	0.6854	+0.000000	yes	0.3391	-0.3463
1	0.6698	0.6698	+0.000000	yes	0.3175	-0.3522
2	0.7035	0.7035	+0.000000	yes	0.3705	-0.3330
3	0.6935	0.6935	+0.000000	yes	0.5005	-0.1930
4	0.6983	n/a	n/a	n/a	n/a	n/a

BWT_routed = +**0.000000**, BWT_unrouted = -0.3061

5.2.2 Seed 2

Patch j	HR_diag	HR_final_routed	routed delta	bit-identical	HR_unrouted	unrouted delta
0	0.6854	0.6854	+0.000000	yes	0.3372	-0.3482
1	0.6676	0.6676	+0.000000	yes	0.3185	-0.3491
2	0.6988	0.6988	+0.000000	yes	0.3652	-0.3335
3	0.6917	0.6917	+0.000000	yes	0.5004	-0.1913
4	0.6978	n/a	n/a	n/a	n/a	n/a

BWT_routed = +**0.000000**, BWT_unrouted = -0.3055

5.2.3 Seed 3

Patch j	HR_diag	HR_final_routed	routed delta	bit-identical	HR_unrouted	unrouted delta
0	0.6854	0.6854	+0.000000	yes	0.3324	-0.3530
1	0.6713	0.6713	+0.000000	yes	0.3226	-0.3488
2	0.6970	0.6970	+0.000000	yes	0.3682	-0.3288
3	0.6962	0.6962	+0.000000	yes	0.5048	-0.1914
4	0.6993	n/a	n/a	n/a	n/a	n/a

BWT_routed = +**0.000000**, BWT_unrouted = -0.3055

5.2.4 Aggregate

Metric	mean	stdev	per-seed values
BWT_routed	+0.000000	0.000000	+0.000000, +0.000000, +0.000000
BWT_unrouted	-0.3057	0.0003	-0.3061, -0.3055, -0.3055

Cross-seed weight-equality assertion: **all passed**, every seed, every $j = 0..3$. HR_diag at $j = 1..4$ equals the plateau column of Table 3.

5.3 Independent check without the dataset

`checkpoint_hashes.csv` lists a SHA-256 per parameter tensor for all 24 paper-4 checkpoints (3,288 rows). It is produced by `gen_checkpoint_hashes.py`, which shares no code with `eval_bwt_posthoc.py`, so the two are independent implementations of the same assertion.

Result, comparing expert j and head j between `*_exp{j}.pt` and `*_exp4.pt`:

Seed	j = 1	j = 2	j = 3
1	identical (32 tensors)	identical (32)	identical (32)
2	identical (32)	identical (32)	identical (32)
3	identical (32)	identical (32)	identical (32)

Nine comparisons, all passing. The 32 tensors per patch group are the five layers' experts (linear1 weight and bias, linear2 weight and bias, norm2 scale and shift: six per layer) plus the patch's output head weight and bias.

We can repeat this with the checkpoints alone: no dataset, no forward pass, no GPU. Reproducing the HR columns additionally requires DOTA-Draft and `eval_bwt_posthoc.py`.

Scope of the hash check. It covers $j = 1, 2, 3$ from the per-seed checkpoints. Patch 0 is trained once and shared across seeds, and its checkpoint is stored outside the per-experience directory, so $j = 0$ bit-identity rests on the `eval_bwt_posthoc.py` assertion above rather than on a second independent check. The $j = 0$ row of the per-seed tables is the same model in all three cases, which is why HR_diag[0] = 0.6854 for every seed.

6 S5: Computational cost

6.1 Wall-clock training time

Total run duration in seconds, W&B field `_runtime`. One full pass over the five-patch stream per row.

Seed	Base (s)	PatchExpertFFN+head (s)	Delta	Host
1	17,448	16,925	-3.00%	GTX 1660 SUPER
2	20,265	20,585	+1.58%	GTX 1660
3	11,872	11,654	-1.84%	RTX 2060 SUPER

This substantiates §6.2.4. The largest absolute difference between arms is 2.997%, and the sign is not consistent: PatchExpertFFN+head is faster on seeds 1 and 3 and slower on seed 2. Training time is cost-neutral.

Compare within a row, not down a column. Each matched pair ran on the same machine, so the within-seed delta is a clean comparison. Absolute times are not comparable across seeds: seed 3 ran on a faster GPU and is roughly 1.7× quicker than seed 2 on both arms. That is hardware, not method. See S0 for the full machine table.

6.2 Runtime effect of trunk freezing

Freezing skips weight-gradient computation and the optimizer step for the trunk, but backpropagation still traverses the full graph to reach the experts, and at this model scale evaluation and data loading dominate. The saving is in samples, not compute, which is a data-efficiency result rather than a throughput one.

6.3 Storage

Per-patch storage fraction ρ = 676,223 / 1,077,247 = 0.628.

T	Sequential FT	Model zoo	PatchExpertFFN+head
0	1.08M	1.08M	1.08M
1	1.08M	2.15M	1.75M
2	1.08M	3.23M	2.43M
3	1.08M	4.31M	3.11M
4	1.08M	5.39M	3.78M

Sequential fine-tuning keeps one evolving model and forgets. The model zoo keeps a full frozen copy per patch. PatchExpertFFN+head keeps one frozen trunk (401,024) plus one expert and head (676,223) per patch, reaching the zoo's zero-forgetting guarantee at 63% of its per-patch cost.

Checkpoint files on disk are 14.43 MB at every T because the reference implementation pre-allocates the whole expert library; see S1. That is an implementation choice, not the deployable cost.

6.4 Inference

Unaffected. Routing is an integer index into the expert library, not a learned gate, so there is no gating network to evaluate and no additional forward-pass cost relative to the shared-weight baseline.

7 S6: Code

Two listings: the algorithm itself, and the routine that verifies the zero-forgetting claim. Both are reproduced from the working code with comments and unused imports removed; each was scanned for identifying strings before inclusion.

The per-experience checkpoints these listings operate on, for both arms, all three seeds and experiences 1 to 4, together with the SHA-256 digest table of S4 and the per-step recovery records of S3, are deposited at `https://anonymous.4open.science/r/supplementary-files-9FDD/`, which allows the S4 verification to be checked without retraining and the S3 curves to be regenerated from source data. The source code is available from the corresponding author on reasonable request and is deposited under a permanent DOI on acceptance.

Listing	Source	Lines
S6.1	`cl_localize/plugins/patch_expert_ffn.py`	155
S6.2	`eval_bwt_posthoc.py`, verification core	99

7.1 S6.1 The algorithm

`PatchExpertFFNPlugin`, an Avalanche `SupervisedPlugin`. It owns the whole training rule and touches nothing else:

- at experience $t \geq 1$, copy expert and head $t-1$ into slot t (the warm

start whose effect is visible at the left of the S3 curves),

- freeze every parameter except expert t and head t,
- route the forward pass through slot t.

Patch identity is an integer index. There is no gate, no router network, and no inference step to decide which expert applies. The `copy_init=False` toggle disables the warm start and exists for the ablation reported as `fwt/hit@10_exp{t}`.

Listing: `code/patch_expert_ffn_plugin.py`

7.2 S6.2 Zero-forgetting verification

The core of `eval_bwt_posthoc.py`: extract expert and head tensors for patch j from two checkpoints, compare them elementwise, and compute routed and unrouted backward transfer over the same evaluation sets.

`tensors_equal` is the function behind the "bit-identical" column of S4. It compares raw tensor contents, not metrics, which is why the routed deltas are exactly `+0.000000` rather than small numbers.

The script reads saved checkpoints and never retrains, so the verification can be repeated at any time by anyone holding the deposit.

Listing: `code/eval_bwt_posthoc_core.py`

7.3 What is not included

The data pipeline, the signal compilers, and the other continual-learning strategies in the codebase are not reproduced here. They are shared infrastructure across a series of studies and are not exercised by the experiments in this manuscript. In particular, no drift-signal compiler is invoked by either arm: routing is on patch identity alone.

```python
from avalanche.training.plugins import SupervisedPlugin


class PatchExpertFFNPlugin(SupervisedPlugin):
    def __init__(self, verbose: bool = True, per_patch_head: bool = False,
                 copy_init: bool = True, eval_datasets=None, k: int = 10,
                 batch_size: int = 1024):
        super().__init__()
        self.verbose = verbose
        self.per_patch_head = per_patch_head
        self.copy_init = copy_init
        self.eval_datasets = eval_datasets or {}
        self.k = k
        self.batch_size = batch_size

    def _resolve_model(self, strategy):
        model = getattr(strategy, "model", None)
        if model is None:
            return None
        if hasattr(model, "set_active_patch") and hasattr(model, "copy_expert_weights"):
            return model
        base = getattr(model, "base_model", None)
        if base is not None and hasattr(base, "set_active_patch") and hasattr(base, "
     copy_expert_weights"):
            return base
        return None

    def _wandb_log(self, payload: dict) -> None:
        try:
            import wandb
        except ImportError:
            return
        if wandb.run is not None:
            wandb.log(payload)

    def _wandb_summary(self, payload: dict) -> None:
        try:
            import wandb
        except ImportError:
            return
        if wandb.run is not None:
            for key, value in payload.items():
                wandb.run.summary[key] = value

    def _log_forward_transfer(self, strategy, m, t: int) -> None:
        ds = self.eval_datasets.get(t)
        if ds is None:
            return
        m.set_active_patch(t - 1)
        if self.per_patch_head:
            m.set_active_head(t - 1)
        from cl_localize.metrics.causal_eval import evaluate_model
        res = evaluate_model(
            strategy.model, ds, strategy.device,
            k=self.k, batch_size=self.batch_size,
            label=f"fwt_exp{t}",
        )
        hr = res.get(f"hit@{self.k}", float("nan"))
        self._wandb_log({f"fwt/hit@{self.k}_exp{t}": hr})
        if self.verbose:
            print(f"[PatchExpertFFN] FWT exp {t}: hit@{self.k}={hr:.4f} "
                f"(patch {t-1}'s state, pre-copy/pre-training).")

    def _log_param_counts(self, m, t: int) -> None:
        trainable = sum(p.numel() for p in m.parameters() if p.requires_grad)
        stored = sum(p.numel() for layer in m.patch_expert_layers
                     for p in layer.patch_experts[t].parameters())
        if self.per_patch_head:
            stored += sum(p.numel() for p in m.patch_heads[t].parameters())
        self._wandb_summary({
            "params/trainable_per_patch": trainable,
            "params/stored_per_patch": stored,
```

```
        })
        if self.verbose:
            print(f"[PatchExpertFFN] params: trainable_per_patch={trainable:,} "
                f"stored_per_patch={stored:,}")

    def before_training_exp(self, strategy, *args, **kwargs):
        m = self._resolve_model(strategy)
        if m is None:
            return
        t = strategy.experience.current_experience
        if t == 0:
            m.set_active_patch(0)
            m.set_active_head(0)
            if self.verbose:
                print("[PatchExpertFFN] train Exp 0: full model "
                    "(expert 0 == backbone FFN, head 0 == backbone final_projection).")
        else:
            self._log_forward_transfer(strategy, m, t)

            if self.copy_init:
                m.copy_expert_weights(src=t - 1, dst=t)
            m.freeze_all_except_experts()
            m.set_trainable_expert(t)
            m.set_active_patch(t)
            if self.per_patch_head:
                if self.copy_init:
                    m.copy_head_weights(src=t - 1, dst=t)
                m.set_trainable_head(t)
                m.set_active_head(t)

            self._log_param_counts(m, t)

            if self.verbose:
                head_msg = f", copied head {t-1} -> {t} (trainable)" if self.per_patch_head else " (
 head pinned at 0)"
                copy_msg = "" if self.copy_init else " [copy_init=False: fresh random init, no warm
 start]"
                print(f"[PatchExpertFFN] train Exp {t}: copied expert {t-1} -> {t}, "
                    f"froze everything but expert {t}{head_msg}{copy_msg}.")

    def after_training_exp(self, strategy, *args, **kwargs):
        m = self._resolve_model(strategy)
        if m is None:
            return
        t = strategy.experience.current_experience
        if t == 0:
            return

        payload = {}
        total_sq = 0.0
        for i, layer in enumerate(m.patch_expert_layers):
            cur = layer.patch_experts[t]
            prev = layer.patch_experts[t - 1]
            sq = 0.0
            for (_, p_cur), (_, p_prev) in zip(cur.named_parameters(), prev.named_parameters()):
                diff = p_cur.detach() - p_prev.detach()
                sq += float((diff * diff).sum())
            l2 = sq ** 0.5
            payload[f"expert_diff/l{i}_l2"] = l2
            total_sq += sq
        payload["expert_diff/total_l2"] = total_sq ** 0.5
        payload["expert_diff/exp"] = t

        if self.per_patch_head:
            h_cur, h_prev = m.patch_heads[t], m.patch_heads[t - 1]
            hsq = float(((h_cur.weight.detach() - h_prev.weight.detach()) ** 2).sum())
            hsq += float(((h_cur.bias.detach() - h_prev.bias.detach()) ** 2).sum())
            payload["head_diff/l2"] = hsq ** 0.5

        self._wandb_log(payload)
        if self.verbose:
            print(f"[PatchExpertFFN] exp {t} drift vs expert {t-1}: "
                f"total_l2={payload['expert_diff/total_l2']:.4f}"
```

```
                + (f", head_l2={payload.get('head_diff/l2', float('nan')):.4f}" if self.per_patch_head
    else ""))

def before_eval_exp(self, strategy, *args, **kwargs):
    m = self._resolve_model(strategy)
    if m is None:
        return
    t = strategy.experience.current_experience
    m.set_active_patch(t)
    if self.per_patch_head:
        m.set_active_head(t)
    if self.verbose:
        head_tag = f", head {t}" if self.per_patch_head else ", head 0"
        print(f"[PatchExpertFFN] eval Exp {t}: expert {t}{head_tag}.")
```

Listing 1: S6.1 PatchExpertFFNPlugin: the training rule

```python
def expert_and_head_tensors(model: PatchExpertFFNBert4Rec, patch_id: int) -> Dict[str, torch.Tensor]:
    sd = model.state_dict()
    out = {}
    for L in range(N_LAYERS):
        prefix = f"patch_expert_layers.{L}.patch_experts.{patch_id}."
        for k, v in sd.items():
            if k.startswith(prefix):
                out[k] = v.detach().clone()
    head_prefix = f"patch_heads.{patch_id}."
    for k, v in sd.items():
        if k.startswith(head_prefix):
            out[k] = v.detach().clone()
    return out


def tensors_equal(a: Dict[str, torch.Tensor], b: Dict[str, torch.Tensor]) -> Dict[str, bool]:
    assert set(a.keys()) == set(b.keys()), (
        f"Tensor key sets differ:\n  only in a: {set(a) - set(b)}\n"
        f"  only in b: {set(b) - set(a)}"
    )
    return {k: bool(torch.equal(a[k], b[k])) for k in a}


def eval_one(model, ds, label: str) -> float:
    res = evaluate_model(model, ds, DEVICE, k=K, batch_size=EVAL_BATCH_SIZE, label=label)
    return res[f"hit@{K}"]


def run_seed(seed: int, test_datasets: List[NewHeroActionBERTDataset]) -> dict:
    print(f"\n{'=' * 60}\nSeed {seed}\n{'=' * 60}")

    hr_diag = [None] * NUM_PATCHES
    diag_expert_head_tensors = {}
    for j in range(NUM_PATCHES):
        m = build_model()
        ckpt_path = ckpt_for(seed, j)
        print(f"[diag] loading {ckpt_path} for patch {j}")
        load_checkpoint(m, ckpt_path)
        m.set_active_patch(j)
        m.set_active_head(j)
        hr_diag[j] = eval_one(m, test_datasets[j], label=f"diag_seed{seed}_exp{j}")
        diag_expert_head_tensors[j] = expert_and_head_tensors(m, j)
        print(f"  HR_diag[{j}] = {hr_diag[j]:.4f}")
        del m

    model_final = build_model()
    exp4_path = ckpt_for(seed, NUM_PATCHES - 1)
    print(f"[routed-final] loading {exp4_path}")
    load_checkpoint(model_final, exp4_path)

    hr_final_routed = [None] * (NUM_PATCHES - 1)
    final_expert_head_tensors = {}
    for j in range(NUM_PATCHES - 1):
        model_final.set_active_patch(j)
        model_final.set_active_head(j)
        hr_final_routed[j] = eval_one(model_final, test_datasets[j], label=f"routed_final_seed{seed}
    _exp{j}")
        final_expert_head_tensors[j] = expert_and_head_tensors(model_final, j)
        print(f"  HR_final_routed[{j}] = {hr_final_routed[j]:.4f}")

    weight_equal = {}
    routed_delta = [None] * (NUM_PATCHES - 1)
    for j in range(NUM_PATCHES - 1):
        eq_map = tensors_equal(diag_expert_head_tensors[j], final_expert_head_tensors[j])
        all_equal = all(eq_map.values())
        weight_equal[j] = all_equal
        routed_delta[j] = hr_final_routed[j] - hr_diag[j]
        if not all_equal:
            bad_keys = [k for k, v in eq_map.items() if not v]
            print(f"  [WARNING] expert/head {j}: weight mismatch on keys: {bad_keys}")
        else:
            print(f"  [OK] expert/head {j}: bit-identical between exp{j} and exp4 checkpoints")
```

```
72
73      bwt_routed = sum(routed_delta) / len(routed_delta)
74
75      hr_unrouted = [None] * (NUM_PATCHES - 1)
76      unrouted_delta = [None] * (NUM_PATCHES - 1)
77      model_final.set_active_patch(NUM_PATCHES - 1)
78      model_final.set_active_head(NUM_PATCHES - 1)
79      for j in range(NUM_PATCHES - 1):
80          hr_unrouted[j] = eval_one(model_final, test_datasets[j], label=f"unrouted_seed{seed}_exp{j}")
81          unrouted_delta[j] = hr_unrouted[j] - hr_diag[j]
82          print(f"  HR_unrouted[{j}] (routed through exp4 expert/head) = {hr_unrouted[j]:.4f}"
83                f"  (delta {unrouted_delta[j]:+.4f})")
84
85      bwt_unrouted = sum(unrouted_delta) / len(unrouted_delta)
86
87      del model_final
88
89      return {
90          "seed": seed,
91          "hr_diag": hr_diag,
92          "hr_final_routed": hr_final_routed,
93          "routed_delta": routed_delta,
94          "bwt_routed": bwt_routed,
95          "weight_equal": weight_equal,
96          "hr_unrouted": hr_unrouted,
97          "unrouted_delta": unrouted_delta,
98          "bwt_unrouted": bwt_unrouted,
99      }
```

Listing 2: S6.2 Zero-forgetting verification, core routine