6 APPENDIX

6.1 ADDITIONAL EXPERIMENTS

Table 3 reports the impact of different memory curation strategies on CIFAR-100-C under episodic Test-Time Adaptation (TTA). Results are averaged over 15 corruption types at severity 5. and Table 4 is the results on the continual version.

We first observe that even uninformed memories (e.g., FIFO, Reservoir) significantly boost performance compared to no memory, showing that simply caching and replaying recent samples benefits all adaptation methods in a dynamic stream. Next, class-guided memories such as PBRS and CSTU — which explicitly enforce class balance — achieve the highest accuracy, highlighting the importance of balanced memory composition.

Our proposed Guided Observational memories (DPP, FPS, FPSD) achieve comparable performance to these class-aware approaches while being more sample-efficient: they selectively admit diverse and informative samples, allowing the model to adapt with fewer updates. This effect is consistent across both episodic and continual evaluation settings, underscoring the robustness of our approach.

Table 3: Impact of memory curation policies on the performance of Test-Time Adaptation methods on CIFAR-100-C. The table shows the mean classification accuracy (%) on a episodic evaluation, averaged over 15 corruptions at severity 5. We compare three adaptation methods (RoTTA, Norm, NOTE) and analyze their sensitivity to the hyperparameter γ when combined with different memory strategies.

Memory	**Rotta**			**Norm**			**Note**		
	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$
NONE	50.01	51.17	56.66	9.12	10.67	20.30	8.65	10.04	18.16
FIFO	60.45	60.95	**63.26**	62.27	62.54	63.74	44.48	44.67	45.30
RESERVOIR	52.01	53.27	52.82	52.99	54.49	53.70	40.52	40.85	40.34
CSTU	59.11	59.22	59.57	59.94	59.68	58.84	45.96	**46.08**	**46.70**
PBRS	**62.30**	**62.46**	63.16	**64.12**	**64.20**	**64.31**	45.02	45.39	45.96
DPP	59.05	59.14	59.54	59.94	59.68	58.84	**46.05**	46.08	46.69
FPS	59.24	59.16	59.59	59.73	59.41	58.69	45.81	45.90	46.62
FPSD	55.62	55.65	57.45	49.34	49.88	51.87	38.97	38.59	38.60

Table 4: Impact of memory curation policies on the performance of Test-Time Adaptation methods on CIFAR-100-C. The table shows the mean classification accuracy (%) on a continual evaluation, averaged over 15 corruptions at severity 5. We compare three adaptation methods (RoTTA, Norm, NOTE) and analyze their sensitivity to the hyperparameter γ when combined with different memory strategies.

Memory	**Rotta**			**Norm**			**Note**		
	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$	$\gamma = 10^{-4}$	$\gamma = 10^{-3}$	$\gamma = 10^{-1}$
NONE	50.27	51.36	57.03	9.12	10.67	20.30	2.94	3.38	6.04
FIFO	60.40	60.93	**63.40**	61.33	61.59	62.86	10.26	10.50	11.96
RESERVOIR	46.25	47.42	46.75	52.30	53.98	53.09	18.15	17.70	17.88
CSTU	58.31	59.29	60.65	56.54	56.47	55.51	20.14	20.99	23.57
PBRS	**61.02**	**61.29**	62.61	**63.02**	**63.02**	**63.25**	17.44	18.91	19.83
DPP	58.38	59.37	60.74	56.54	56.47	55.51	20.61	20.67	24.40
FPS	58.50	59.32	60.89	56.38	56.20	55.29	20.66	22.06	26.95
FPSD	53.22	53.55	56.27	42.21	42.29	42.89	**27.65**	**26.50**	**28.24**

The following table is the episodic version of 2

6.2 CODE IMPLEMENTATION

6.2.1 FARTHEST POINT SAMPLING (FPS)

Table 5: Comparative analysis of memory curation policies under severe dataset shift. The table reports mean classification accuracy (%) on **CIFAR-10** → **CIFAR-10C**, averaged over 15 common corruptions at the highest severity level (5). All evaluations are conducted in an **episodic** correlatively sampled stream with $\gamma = 10^{-1}$.

Policy	Memory \ Method	Source	CoTTA	EATA	Energy	ETA	Norm	NOTE	PL	RoTTA	SAR	SHOT	TENT
Uninformed	NONE	56.49	23.54	**24.72**	**24.72**	**24.72**	24.72	25.75	16.46	47.53	24.75	23.77	**24.72**
	FIFO	-	**24.72**	**24.72**	**24.72**	**24.72**	65.07	73.15	16.07	53.92	24.74	23.81	**24.72**
	Reservoir	-	**24.72**	**24.72**	**24.72**	**24.72**	63.41	70.48	44.80	62.35	24.73	24.26	**24.72**
Class-Guided	PBRS	-	**24.72**	**24.72**	**24.72**	**24.72**	**76.71**	74.03	**51.28**	74.48	**24.75**	23.89	**24.72**
	CSTU	-	**24.72**	**24.72**	**24.72**	**24.72**	75.72	73.77	41.26	73.49	24.72	23.76	**24.72**
Guided Observational	DPP (Ours)	-	**24.72**	**24.72**	**24.72**	**24.72**	75.35	74.39	20.05	75.34	24.72	24.29	**24.72**
	FPS (Ours)	-	**24.72**	**24.72**	**24.72**	**24.72**	76.64	73.43	23.84	**75.97**	24.72	**24.45**	**24.72**
	FPSD (Ours)	-	**24.72**	**24.72**	**24.72**	**24.72**	76.43	**74.93**	48.60	71.47	24.72	23.13	**24.72**

```python


class FPS(BaseMemory):
    def __init__(self, capacity, num_class, min_dist=0.5, lambda_t=1.0, lambda_u=1.0):
        super().__init__(capacity, num_class, lambda_t, lambda_u)
        self.min_dist = min_dist

    def add_instance(self, instance, is_update_feature=False):
        """
        Instance format: (x, prediction, uncertainty, feature)
        """
        new_item = instance
        uncertainty = new_item.uncertainty
        feature = new_item.feature
        prediction = new_item.cls
        new_score = self.heuristic_score(0, uncertainty)


        if self.remove_instance(new_item) and not is_update_feature:
            if self.adds_sufficient_diversity(feature, prediction):
                self.data[prediction].append(new_item)
        elif is_update_feature:
            self.data[prediction].append(new_item)
        self.add_age()

    def adds_sufficient_diversity(self, feature, cls):
        """Ensure new feature is not a near-duplicate within its class."""
        class_list = self.data[cls]
        if not class_list:
            return True
        features = np.array([item.feature for item in class_list])
        dists = cdist(features, feature.reshape(1, -1), metric="euclidean").flatten()
        return np.min(dists) > self.min_dist

    def remove_instance(self, new_item):
        class_list = self.data[new_item.cls]
        if len(class_list) < self.per_class:
            if self.get_occupancy() < self.capacity:
                return True
            else:
                return self.remove_from_classes(self.get_majority_classes(), new_item)
        return self.remove_from_classes([new_item.cls], new_item)


```

```python

    def remove_from_classes(self, classes, new_item):
        """Remove one instance from given classes to make room, preserving diversity.
           """
        min_class, min_index, min_score, min_dist = None, None, None, None
        score_base = self.heuristic_score(new_item.age, new_item.uncertainty)

        for cls in classes:
            if len(self.data[cls]) == 0:
                continue

            items = self.data[cls]
            # Stack features of existing items (float32 to save memory)
            features = np.stack([item.feature.astype(np.float32) for item in items])
            new_f = new_item.feature.astype(np.float32)[None]

            # Distances new_item - existing items
            dists = cdist(new_f, features, metric="euclidean")[0]
            nearest_idx = np.argmin(dists)
            nearest_dist = dists[nearest_idx]

            # Compare score between candidate and nearest existing item
            existing_item = items[nearest_idx]
            score_existing = self.heuristic_score(existing_item.age, existing_item.
                uncertainty)
            score_new = score_base

            # Remove whichever is "worse"
            if score_existing >= score_new:
                remove_idx, remove_score = nearest_idx, score_existing
            else:
                remove_idx, remove_score = None, score_new  # means: reject new item

            # Keep track of best candidate removal across classes
            if remove_idx is not None:
                if min_dist is None or nearest_dist < min_dist:
                    min_class, min_index, min_score, min_dist = cls, remove_idx,
                        remove_score, nearest_dist

        # Execute removal
        if min_class is not None:
            self.data[min_class].pop(min_index)
            return True

        return False

    def timeliness_reweighting(self, ages):
        if isinstance(ages, list):
            ages = torch.tensor(ages).float().cuda()
        return torch.exp(-ages) / (1 + torch.exp(-ages))

    def update_features(self, cfg, model):

        if cfg.MEMORY.IF_DYNAMIC:
            batch, _ = self.get_memory()
            self.clear()
            batch = torch.stack(batch)
            device = next(model.parameters()).device
            batch = batch.to(device, non_blocking=True)
            with torch.no_grad():
                out = model(batch, if_adapt=False)
                probs = F.softmax(out, dim=1)

            # move features to CPU
            features = out.cpu().numpy()
```

```
            uncertainties = -torch.max(probs, dim=1).values.cpu().numpy()

            # assign back
            for i in range(len(batch)):
                feature = features[i]
                uncertainty = uncertainties[i]
                x = batch[i].cpu()
                prediction = torch.argmax(probs[i]).item()
                instance = (x, prediction, uncertainty, feature)
                self.add_instance(instance, is_update_feature=True)
```

Listing 1: Python implementation of the FPS memory policy.

6.2.2 Determinantal Point Process (DPP)

```

class DPP(BaseMemory):
    def __init__(self, capacity, num_class=2, sample_check=50):
        super().__init__(capacity, num_class)
        self.sample_check = sample_check

    def adds_diversity(self, new_item, min_cos_dist=0.1):
        class_list = self.data[new_item.cls]
        if not class_list:
            return True
        sample = random.sample(class_list, min(self.sample_check, len(class_list)))
        nf = new_item.feature / (np.linalg.norm(new_item.feature) + 1e-8)
        feats_n =  np.array([it.feature / (np.linalg.norm(it.feature) + 1e-8) for it
            in sample])
        sims = feats_n @ nf
        return np.max(sims) < (1 - min_cos_dist)

    def remove_from_classes(self, classes, new_item):
        min_class = None
        min_index = None
        min_score = None
        score_base = self.heuristic_score(new_item.age, new_item.uncertainty)
        for cls in classes:
            if not self.data[cls]:
                continue

            items = self.data[cls] + [new_item]
            feats = np.array([it.feature / (np.linalg.norm(it.feature) + 1e-8) for it
                in items])
            sims = feats @ feats.T
            np.fill_diagonal(sims, -1)
            redundant_idx = np.unravel_index(np.argmax(sims), sims.shape)[0]
            if redundant_idx < len(self.data[cls]):
                item = self.data[cls][redundant_idx]
                score = self.heuristic_score(item.age, item.uncertainty)
                if min_score is None or score >= min_score:
                    min_class, min_index, min_score = cls, redundant_idx, score
        if min_class is not None and min_score >= score_base:
            self.data[min_class].pop(min_index)
            return True
        return False

    def remove_instance(self, new_item):
        cls = new_item.cls
        if len(self.data[cls]) < self.per_class:
            if self.get_occupancy() < self.capacity:
                return True
            else:
                majority = np.argmax([len(x) for x in self.data])
```

```
                return self.remove_from_classes([majority], new_item)
        else:
            return self.remove_from_classes([cls], new_item)

    def add_instance(self, new_item):
        if self.remove_instance(new_item):
            if self.adds_diversity(new_item):
                self.data[new_item.cls].append(new_item)
        self.add_age()
```

Listing 2: Python implementation of the DPP memory policy.