7 Appendix

7.1 Proof of Lemma 1

Prior works [9, 10] have used the high-level notion that staying "close" to the pre-trained model can help maintain its robustness capability to justify using projection for fine-tuning. However, there is more than one way to encourage this, for example, regularization [28], a small learning rate [7], and projection [10]. It is not immediately clear why projection is a principled approach. To understand FTP's capability to maintain the pre-trained mode's robustness, we first propose to establish a connection between Lipschitz continuity, a commonly used measure of robustness [11, 12, 13], and fine-tuning through a new definition of *difference function* in the Lemma 1.

Proof. We first expand the difference functions in Eq. 6, i.e. plugging in $\Delta h(\cdot) = h_f(\cdot) - h_0(\cdot)$,

$$\begin{aligned} &\|\Delta h(\mathbf{x}) - \Delta h(\mathbf{x}')\|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \quad \forall (\mathbf{x}, \mathbf{x}') \in \mathbb{R}^m \\ \rightarrow &\| \left(h_f(\mathbf{x}) - h_0(\mathbf{x})\right) - \left(h_f(\mathbf{x}') - h_0(\mathbf{x}')\right) \|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \\ \rightarrow &\| \left(h_f(\mathbf{x}) - h_f(\mathbf{x}')\right) - \left(h_0(\mathbf{x}) - h_0(\mathbf{x}')\right) \|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \end{aligned} \tag{8}$$

Then we apply the reverse triangular inequality to the left-hand side of Eq. 8.

$$\left| \|h_f(\mathbf{x}) - h_f(\mathbf{x}')\|_{\mathrm{h}} - \|h_0(\mathbf{x}) - h_0(\mathbf{x}')\|_{\mathrm{h}} \right| \le \| \left(h_f(\mathbf{x}) - h_f(\mathbf{x}')\right) - \left(h_0(\mathbf{x}) - h_0(\mathbf{x}')\right) \|_{\mathrm{h}}$$

Therefore, we have,

$$\begin{aligned} &\|h_f(\mathbf{x}) - h_f(\mathbf{x}')\|_{\mathrm{h}} - \|h_0(\mathbf{x}) - h_0(\mathbf{x}')\|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \\ &\qquad \rightarrow \|h_f(\mathbf{x}) - h_f(\mathbf{x}')\|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} + \|h_0(\mathbf{x}) - h_0(\mathbf{x}')\|_{\mathrm{h}} \end{aligned} \tag{9}$$

Assuming that the pre-trained model h_0 is L_0-Lipschitz, we know that $\|h_0(\mathbf{x}) - h_0(\mathbf{x}')\|_{\mathrm{h}} \le L_0 \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}}, \quad \forall (\mathbf{x}, \mathbf{x}') \in \mathbb{R}^m$. Plug this into Eq. 9,

$$\|h_f(\mathbf{x}) - h_f(\mathbf{x}')\|_{\mathrm{h}} \le (L_d + L_0) \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \tag{10}$$

□

7.2 Proof of Lemma 2

In the previous section, we established a connection between the robustness of a fine-tuned model $h_f(\cdot)$ and its difference function $\Delta h(\cdot)$. Naturally, if we can limit the Lipschitz constant L_d of the difference function, we can maintain the robustness of the pre-trained model. In this section, we show *projection* as an effective method to enforce the L_d-Lipschitz condition in Eq 6.

Proof. **Linear Operators.** A neural network is composed of linear operators with connecting non-linear activations. Following prior works [9, 10], we analyze the linear operators[3]: $h(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}, \mathbf{W} \in \mathbb{R}^{n \times m}, \mathbf{b} \in \mathbb{R}^n$. Let's define $h_f(\mathbf{x}) = \mathbf{W}_f \mathbf{x} + \mathbf{b}_f$ and $h_0(\mathbf{x}) = \mathbf{W}_0 \mathbf{x} + \mathbf{b}_0$, and plug them in Eq. 6.

$$\|(\mathbf{W}_f - \mathbf{W}_0)(\mathbf{x} - \mathbf{x}')\|_{\mathrm{h}} \le L_d \|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}} \quad \forall (\mathbf{x}, \mathbf{x}') \in \mathbb{R}^m.$$

Rearranging the above equation gives us an upper bound on L_d,

$$L_d = \sup \left\{ \frac{\|(\mathbf{W}_f - \mathbf{W}_0)(\mathbf{x} - \mathbf{x}')\|_{\mathrm{h}}}{\|\mathbf{x} - \mathbf{x}'\|_{\mathrm{x}}} \quad \forall (\mathbf{x}, \mathbf{x}') \in \mathbb{R}^m \right\}. \tag{11}$$

Matrix Norms. Eq. 11 matches the definition of a matrix norm for a matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$: $\|\mathbf{W}\|_{\mathrm{h,x}} = \sup \left\{ \frac{\|\mathbf{W}x\|_{\mathrm{h}}}{\|x\|_{\mathrm{x}}}, \quad \forall x \in \mathbb{R}^n \quad \text{with} \quad x \neq 0 \right\}$. *Therefore, to minimize L_d in Eq. 6, we just need to minimize the matrix norm $\|\mathbf{W}_f - \mathbf{W}_0\|_{\mathrm{h,x}}$.* Note that different vector norm combinations ($\|\cdot\|_{\mathrm{h}}$ and $\|\cdot\|_{\mathrm{x}}$) will lead to a different matrix norm $\|\cdot\|_{\mathrm{h,x}}$. Certain vector norm combinations have a closed-form matrix norm while the majority do not. Following prior works [9, 10], we use Maximum Absolute Row Sum (MARS) matrix norm, which is characterized by l_∞ vector norms in both domains.

---

[3] Convolutional layers can be also written in the matrix multiplication form using Toeplitz matrix.

Specifically, given a desired constraint L_d, we want $\|\mathbf{W}_f - \mathbf{W}_0\|_{\infty,\infty} \leq L_d$. Per the definition of the MARS matrix norm, which is the largest l_1 norm of each row of a matrix, the inequality can be equivalently enforced for each row independently, i.e.,

$$\|\mathbf{W}_f - \mathbf{W}_0\|_{\infty,\infty} \leq L_d \iff \|\mathbf{w}_f^i - \mathbf{w}_0^i\|_1 \leq L_d, \quad \forall i \in \{1, ..., n\}. \tag{12}$$

where $\mathbf{w}^i$ denotes the i-th row of the matrix $\mathbf{W}$.

Projection. To ensure the inequality in Eq. 12, we can *project* $\mathbf{W}_f$ towards $\mathbf{W}_0$ using the following projection equation. For each row $\mathbf{w}^i$ in a matrix $\mathbf{W}$, the projected weight $\tilde{\mathbf{w}}_p^i$ is calculated by

$$\mathbf{w}_p^i = \min\left(1, \frac{\gamma}{\|\mathbf{w}_f^i - \mathbf{w}_0^i\|_1}\right)(\mathbf{w}_f^i - \mathbf{w}_0^i) + \mathbf{w}_0^i.$$

It is easy to check that $\mathbf{w}_p^i$ satisfies Eq. 12, i.e., $\|\mathbf{w}_p^i - \mathbf{w}_0^i\|_1 \leq L_d$ if $0 \leq \gamma \leq L_d$. $\square$

Lipschitz Bound. Since a neural network is a composition of linear operators and non-linear activations, by the composition rule of the Lipschitz functions, an upper bound of the entire network is just the product of the Lipschitz constant for each linear operator and non-linear activations, where most non-linear activations are 1-Lipschitz [13]. However, the Lipschitz bound obtained by using the composition rule is not a tight bound on the entire network. While it is an active research area to find tighter bounds for neural networks without relying on the layer-wise composition rule [52, 12], the layer-wise approach is particularly suitable for connecting the fine-tuning process and Lipschitz continuity because it leads to layer-wise regularization techniques as we demonstrated above.

7.3 FTP: Additional Discussion

In the main paper Sec. 3.2, we described the algorithmic difference between TPGM and FTP. However, there is an implicit assumption made as a result of the difference. We now discuss the implication of it. After obtaining the updated constraints γ_t in Eq. 5, if the algorithm were to follow TPGM, the next step would be applying the updated constraints to *re-calculate* the previous model $\mathbf{W}_{t-1}$. However, instead of rolling back, FTP applies the updated constraints directly to the current unconstrained model $\tilde{\mathbf{W}}_t$. This step assumes *smoothness* in the update of γ_t, i.e., the γ_t does not change drastically in consecutive steps. The assumption is valid since γ_t is updated by *AdampUpdate* (Alg. 2 below) which uses a moving average update with a momentum of 0.9. So the change of γ_t is very smooth because of the high discount factor of 0.9. Importantly, we have **re-used** the same gradient $\mathbf{g}_t$ available for computing the current unconstrained model $\tilde{\mathbf{W}}_t$. This is the key to saving computation because calculating the forward and backward pass through the model is the main computation bottleneck in TPGM because it requires a separate training loop as a result of "rolling back".

Algorithm 2 AdampUpdate: AdamUpdate implements one step update of Adam [30]

```
Require: $\gamma_{t-1}, \nabla\gamma_t, t$                                         ▷ Input
Require: $\mu \leftarrow 1e-2, (\beta_1, \beta_2) \leftarrow (0.9, 0.999), \epsilon \leftarrow 1e-8$   ▷ Fixed parameters for AdamUpdate
Require: $m_1 \leftarrow 0$                                                   ▷ Initialize 1st moment vector
Require: $v_1 \leftarrow 0$                                                   ▷ Initialize 2nd moment vector
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla\gamma_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)\nabla\gamma_t^2$
  $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
  $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
  $\gamma_t \leftarrow \gamma_{t-1} - \mu\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$
```

7.4 Image Classification Experiments Details and Additional Results

In Sec. 4.1.1, we presented image classification results on DomainNet-100% data (111,307 images). Now we further present results using only 10% (11,031 images) of the training data in Tab. 6. In this case, projection-based methods, TPGM and FTP achieved the best performance, demonstrating their regularization capability under low-label conditions. Similar to findings in the main paper, FTP is

Table 6: **DomainNet Results using CLIP pre-trained ResNet50 with 10% Real Data.** FFTP achieves competitive OOD performance and is much faster than prior work TPGM [10] by 37%.

	ID	OOD				Statistics			
	Real	Sketch	Painting	Infograph	Clipart	OOD Avg.	ID Δ (%)	OOD Δ (%)	Time (s/it)↓
Vanilla FT	57.35 (1.43)	17.48 (0.68)	25.60 (0.70)	10.30 (1.57)	23.01 (0.65)	19.10	0.00	0.00	0.54
LP	47.19 (0.93)	17.81 (0.25)	22.71 (2.08)	17.13 (0.75)	17.59 (0.69)	18.81	-17.71	-1.52	0.13
PF [25]	71.04 (0.91)	27.87 (1.04)	38.31(1.05)	19.85 (0.70)	**33.92** (1.53)	29.99	23.86	57.01	0.31
L2-SP [28]	61.41 (0.92)	22.61 (0.52)	30.48 (0.42)	12.28 (0.50)	26.59 (0.57)	22.99	7.08	20.37	0.61
MARS-SP [9]	52.53 (0.84)	15.34 (0.54)	21.57 (0.45)	8.49 (0.60)	19.96 (0.01)	16.34	-8.41	-14.44	0.60
LP-FT [7]	64.11 (0.78)	20.54 (0.27)	30.89 (0.41)	13.58 (0.63)	29.55 (0.82)	23.64	11.78	23.77	-
TPGM [10]	**73.16** (1.27)	**29.88** (0.81)	36.80 (1.42)	19.72 (0.12)	35.28 (0.74)	**30.42**	**27.56**	**59.27**	1.10
FTP	72.89 (0.34)	27.44 (0.13)	**38.11** (0.26)	**20.20** (0.26)	33.58 (0.49)	29.83	27.10	56.19	0.69

up to 37% faster than TPGM during training. Next, we describe the hyper-parameters for all image classification experiments in Sec. 4.1 and above.

DomainNet. We use the released code from the prior work, TPGM [10] to train our FTP model. Therefore, we directly use the reported results from TPGM for competing methods. For FTP, we apply constraints to all trainable layers except for the last linear classification layers. For all experiments, we use SGD as the base optimizer with a weight decay of $5e-4$. For DomainNet-100% and DomainNet-10% experiments, we train models for 50 and 150 epochs respectively with a batch size of 256. We sweep a range of learning rates and use the validation split to determine the best learning rate for FTP for each experiment. Here is the list of best-validated learning rates for all DomainNet experiments.

- DomainNet-100% MOCO-V3 ResNet50 (Tab. 1): $1e-2$
- DomainNet-100% CLIP ResNet50 (Tab. 2): $1e-2$
- DomainNet-10% CLIP ResNet50 (Tab. 6): $1e-1$

Note that we use the default $\kappa = 1$ for all these experiments. Every DomainNet experiment was conducted using 4 RTX 2080 GPUs.

ImageNet. For ImageNet experiments (Tab. 3, Fig. 3), we use a CLIP pre-trained ViT-Base [4]. Unlike the DomainNet experiments, we also initialize the last linear layer with zero-shot weights extracted from a CLIP text encoder, following the prior work WISE [8]. Therefore, FTP is applied to all trainable layers including the last linear layer. Training Transformers have been well-studied with abundant regularization and augmentation techniques. To obtain the best fine-tuning performance, we follow the public code base of DEIT [31] to fine-tune all methods. Specifically, we use weight-decay (0.1), drop-path (0.2) [32], label-smoothing (0.1) [33], Mixup (0.8) [34] and Cutmix (1.0) [35]. One exception is Linear-Probing (LP), where we do not use any of the above augmentations because they have been shown to degrade linear probing performance [3, 1]. We train all methods using AdamW [14] as the base optimizer with a weight decay of 0.1, cosine learning rate schedule, and a batch size of 256 for 30 epochs. We also sweep relevant hyper-parameters for each method and document them below.

- FT: learning rate $2e-5$
- LP: learning rate $5e-3$
- LP-FT: learning rate $2e-5$. We take the best LP model (trained for 30 epochs) and then fine-tune it for another 15 epochs with the learning rate specified above.
- L2-SP: learning rate $2e-5$, regularization hyper-parameter $1e-5$.
- FTP: learning rate $3e-5$, regularization hyper-parameter default $\kappa = 1$.

Every ImageNet classification experiment was conducted on 2 A40 GPUs.

7.5 PASCAL Dense Vision Task Experiments Details and Additional Results

In Sec. 4.2, we presented results on semantic segmentation. In this section, we provide the additional results on semantic segmentation and surface normal estimation in Tab, 7 and Tab. 8. FTP achieves the best ID and OOD performance with significantly improved computation efficiency over TPGM [10]. Next, we will give more details on implementation.

Table 7: Pascal Human Parts Segmentation Results using SWIN-Tiny transformers pre-trained on ImageNet21K. Performance is measured by mIoU↑. FTP achieves the best OOD performance and is much faster than prior work TPGM [10] by **34**%.

	ID	OOD				Statistics			
	Clean	Fog	Defocus	Gaussian	Brightness	OOD Avg.	ID Δ (%)	OOD Δ (%)	Time (s/it)↓
Vanilla FT	62.61 (0.31)	57.50 (0.73)	40.76 (0.19)	30.64 (0.88)	57.47 (0.33)	46.59	0.00	0.00	0.280
Adapter	60.84 (1.27)	57.11 (0.39)	45.03 (3.96)	33.12 (1.92)	57.25 (0.68)	48.13	-2.81	3.30	0.221
BitFit	59.06 (0.97)	55.66 (1.36)	**45.81** (1.27)	32.18 (2.59)	55.89 (0.97)	47.39	-5.67	1.70	0.235
L2-SP	62.26 (3.17)	58.46 (2.83)	45.35 (1.30)	34.36 (2.79)	58.40 (2.52)	49.14	-0.56	5.47	0.336
MARS-SP	62.92 (0.94)	58.04 (1.75)	42.51 (1.72)	32.66 (2.53)	58.33 (1.15)	47.89	0.50	2.77	0.308
LLRD	64.37 (1.80)	60.10 (2.58)	44.61 (1.95)	36.90 (4.84)	59.84 (2.06)	50.36	2.81	8.09	0.278
TPGM	63.29 (1.72)	60.16 (1.44)	46.91 (1.78)	37.30 (2.60)	59.81 (1.00)	51.04	1.10	9.55	0.602
FTP	**65.50** (0.17)	**61.73** (0.36)	44.97 (0.70)	**40.55** (1.71)	**61.23** (0.12)	**52.12**	**4.63**	**11.86**	0.397

Table 8: Pascal surface normal Results using SWIN-Tiny transformers pre-trained on ImageNet21K. Performance is measured by RMSE↓. FTP achieves the best OOD performance and is much faster than prior work TPGM [10] by 35%.

	ID	OOD				Statistics			
	Clean	Fog	Defocus	Gaussian	Brightness	OOD Avg.	ID Δ (%)	OOD Δ (%)	Time (s/it)↓
Vanilla FT	18.98 (0.05)	22.25 (0.08)	23.51 (0.06)	27.33 (0.20)	20.83 (0.06)	23.48	0.00	0.00	0.288
Adapter	18.19 (0.05)	20.15 (0.04)	21.46 (0.02)	23.90 (0.14)	19.23 (0.06)	21.19	-4.15	-9.77	0.229
BitFit	20.01 (0.05)	21.93 (0.03)	23.95 (0.12)	26.92 (0.18)	21.28 (0.05)	23.52	5.43	0.17	0.240
L2-SP	16.51 (0.04)	19.26 (0.13)	20.49 (0.11)	24.46 (0.29)	18.08 (0.04)	20.57	-13.01	-12.38	0.343
MARS-SP	19.01 (0.04)	22.15 (0.13)	23.69 (0.11)	27.53 (0.29)	20.86 (0.04)	23.56	0.18	0.32	0.313
LLRD	15.54 (0.08)	18.31 (0.03)	**20.01** (0.20)	26.47 (1.45)	17.36 (0.07)	20.54	-18.11	-12.54	0.279
TPGM	18.17 (0.02)	19.74 (0.04)	21.00 (0.15)	**23.53** (0.27)	19.02 (0.03)	20.82	-4.24	-11.32	0.616
FTP	**15.51** (0.10)	**18.19** (0.09)	**20.01** (0.21)	26.39(0.78)	**17.32** (0.10)	**20.48**	**-18.30**	**-12.79**	0.403

Following prior works [42], we use a combination of Swin-Tiny Transformer [44] encoder and Segformer [45] decoder. The decoder is customized to allow different output formats. Only the Swin encoder is initialized with pre-trained weights (pre-trained on ImageNet-22k). Therefore, we only apply FTP to the encoder. For all methods, we use Adam as the base optimizer with a weight decay of $1e-4$ and a learning rate of $1e-4$ for 60 epochs. For methods with regularization hyper-parameters, we sweep a range of values and report the best one. We provide Tab. 9 for reference.

Table 9: Hyper-parameters for PASCAL Dense Vision Tasks Experiments.

	Semseg	Human Parts	Surface Normal
L2-SP	5e-4	1e-4	1e-4
LLRD	0.65	0.45	0.65
MARS-SP	4	8	4
FTP	1.0	0.0	0.0

To test OOD robustness on the PASCAL-Context benchmark, we apply natural corruptions to the original clean images. Specifically, we select four types of corruptions from the popular benchmark [43], each of which is sampled from a main category: noise, blur, weather, and digital. Each corruption has five levels of severity. We report the average values over the five severity in our paper. Here, we also provide a detailed breakdown for each level of corruption in Fig. 5. Every PASCAL experiment was conducted on a single RTX 2080 GPU.

7.6 Continual Learning Experiments Details and Additional Results

In this section, we provide a brief overview of the settings in continual learning (CL). In CL, a model θ is trained on a sequence of task $n \in \{1, ..., N\}$. Each task has a non-overlapping set of class labels T_n, and we denote the number of classes as $|T_n|$. For ImageNet-R, we split the 200 classes into 10 tasks with 20 labels each, i.e., $N = 10, |T_n| = 20$. Our experiments belong to the class-incremental category in CL. With each new task, the final linear classifier layer is expanded with randomly initialized weights. We denote $\theta_{i,1:n}$ as the model that has been trained on i tasks and the classifier has all classes up to and including the n-th task ($i \geq n$).

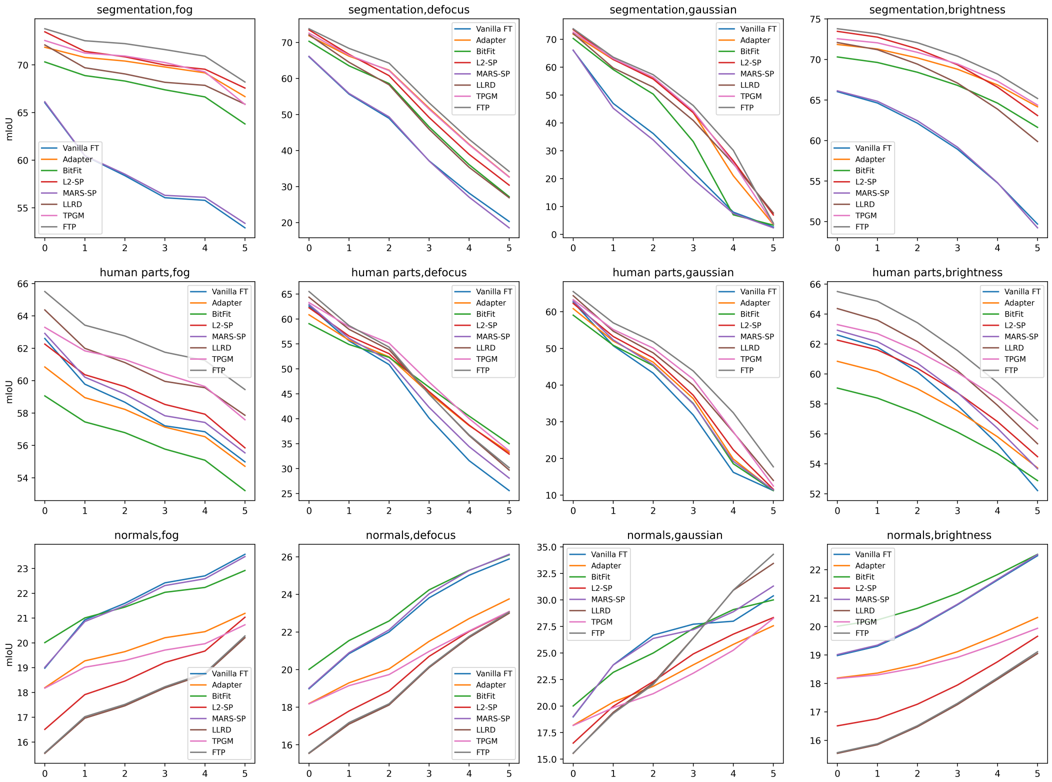

Figure 5: Performance Breakdown for each Level of Corruption on PASCAL-Context Vision Tasks.

To measure global performance, we first define the *global task accuracy* $A_{1:N}$ as,

$$A_{1:N} = \frac{1}{|D^{\text{test}}|} \sum_{(x,y)\in D^{\text{test}}} \mathbf{I}(\hat{y}(x, \theta_{N,1:N}) = y).$$

where D_{test} is the test dataset which has data from all N tasks and $\hat{y}(x, \theta)$ denotes the predicted class from the model with weights θ. Then we define the *global forgetting* F_N [53] as,

$$F_N = \frac{1}{N-1} \sum_{i=2}^{N} \sum_{n=1}^{i-1} \frac{|T_N|}{T_{1:n}} (R_{n,n} - R_{i,n})$$

where,

$$R_{i,n} = \frac{1}{|D_n^{\text{test}}|} \sum_{(x,y)\in D_n^{\text{test}}} \mathbf{I}(\hat{y}(x, \theta_{i,1:n}) = y).$$

Following the prior work [51], all experiments in Tab. 5 use a ViT-Base pre-trained on ImageNet. We tune FTP with the code provided by the authors and directly compare it to the results from the prior work. Specifically, all methods use Adam as the base optimizer with no weight decay and a batch size of 128. All results are averaged over 3 random seed trials where the class allocation to each task is shuffled. For FTP, we train the model for 25 epochs with an initial learning rate of $5e-4$ and a cosine learning rate schedule. For all methods, we freeze the majority of the backbones and only fine-tune the QKV attention layers in the ViT. Please refer to the prior work for a more detailed description of the compared methods. Every CL experiment was conducted on 4 RTX2080 GPUs.

7.7 Pytorch Code Example of FTP

Here is an example of using SGDP (SGD+FTP) in Pytorch format. SGDP requires the common arguments for initializing an SGD optimizer class in Pytorch with two additional inputs: k and exclude_set. k is the hyper-parameter for positive gradient annealing (Sec. 3.2) and exclude_set contains the set of the names of parameters to be excluded from the projection operation. A complete demonstration of image classification is provided in the supplementary. You should be able to reproduce FTP results in Tab. 1 and Tab. 2.

```python
from FTP import SGDP

# Parameters to be optimized
params_to_opt = [x[1] for x in model.named_parameters()]
# Names of parameters to be optimized
params_to_opt_name = [x[0] for x in model.named_parameters()]
# Copy the initial parameters as the anchor
params_anchor = copy.deepcopy(params_to_opt)
# Set up the parameter groups
param_group = [{"params":params_to_opt,
                "pre": params_anchor,
                "name": params_to_opt_name}]
# Set up the optimization hyper-parameters
optimizer_params = {
        "lr": 1e-2,
        "weight_decay": 5.0e-4,
        "momentum": 0.9,
        "nesterov": True,
        "k":1.0,
        "exclude_set":{’module.head.weight’,’module.head.bias’}
    }
optimizer = SGDP(param_group,**optimizer_params)
```