# Interpretable Image Classification via Non-parametric Part Prototype Learning

## Supplementary Material

This document provides more implementation and experimental details of our framework, as well as more quantitative and qualitative results:

- Section A provides more implementation details
- Section B describes additional details on experiments
- Section C provides more experimental results
- Section D provides additional visualizations in comparison with previous works

## A. Implementation Details

During the first stage of training, given the patch tokens $\mathbf{f} \in \mathbb{R}^{Bhw \times D}$ that encodes a batch of images, the foreground extraction module $\phi$ estimates a binary mask $\mathcal{M} \in \{0,1\}^{Bhw}$ where a value of 1 indicates that the corresponding patch token is in the foreground of the input image it belongs to. With this, the patch tokens $\mathbf{f}^c$ that belongs to each class $c$ in a batch of training samples can be defined as:

$$\mathbf{f}^c := \{\mathbf{f}_i \mid \mathcal{M}_i = 1, \mathcal{Y}_i = c\},$$

where $\mathcal{Y} \in \{c_1, c_2, ...\}^{Bhw}$ denotes the class label of the input image that each patch token belongs to. The foreground extractor $\phi$ can be implemented using various off-the-shelf methods, such as PaPr [26], Principle Component Analysis [27] and Attention Rollout [46]. We ablate these methods in Table 3. In addition, we adopt cosine similarity to compute similarity $\mathbf{S}$ between the patch tokens $\mathbf{f}$ and prototypes $\mathbf{P}$:

$$\mathbf{S} = \frac{\mathbf{f}^\top \mathbf{P}}{\|\mathbf{f}\|_2 \|\mathbf{P}\|_2}.$$

During the first stage of training, we initialize the prototypes from $\mathcal{N}(0.0, 0.02)$ and the modulation weights with $w_k^c = 0.2$. We provide an overview of our training procedure in Algorithm 1. When adapting to concept learning, we construct the concept bottleneck layer by replacing the modulation weights $w$ with two fully connected layers, where the output dimension of the first layer equal to the number of concepts associated with all classes.

**Distinctiveness and Comprehensiveness Scores.** The calculation of Comprehensiveness Score requires the ground truth foreground masks of test images, while the calculation of Distinctiveness Score is label-free. We provide Python-like pseudo-code for calculating these two scores in Algorithm 2 and Algorithm 3.

**Algorithm 1** Prototype Learning and Backbone Fine-tuning

**Require:** Training set $\mathcal{X}$, $nEpochs$, ViT backbone $\mathcal{F}$

Initialize prototypes $\mathbf{P}$ and modulation vector $w$ with $\mathbf{P} \sim \mathcal{N}(0.0, 0.02)$ and $w_k^c = 0.2$
**for** $t \in \{1, \ldots, nEpochs\}$ **do**
  Randomly split $\mathcal{X}$ into mini-batches
  **for** $(\mathbf{X}, \mathbf{y}) \in \{\mathcal{X}_1, \ldots\}$ **do**
    $\mathbf{f} = \text{reshape}(\mathcal{F}(\mathbf{X})) \in \mathbb{R}^{Bhw \times D}$
    $\mathcal{M} = \phi(\mathbf{f}, \mathbf{X}) \in \{0,1\}^{Bhw}$
    $\mathbf{S} = \text{reshape}(\text{Sim}(\mathbf{f}, \mathbf{P})) \in \mathbb{R}^{Bhw \times N \times K}$
    **for** $c \in \text{unique}(\mathbf{y})$ **do**
      Get $\mathbf{S}^c$ and $\mathbf{f}^c$ by indexing $\mathbf{S}$, $\mathbf{f}$ with $\mathcal{M}$, $\mathbf{y}$
      $\mathbf{A}^{c,*} = \text{sinkhorn\_knopp}(\mathbf{S}^c)$
      **if** *first training stage* **then**
        $\mathbf{P}^c \leftarrow \beta \mathbf{P}^c + (1-\beta)(\mathbf{A}^{c,*})^\top \mathbf{f}^c$
      **end if**
    **end for**
    $g = \text{MaxPool}(\text{reshape}(\mathbf{S})) \in \mathbb{R}^{N \times K}$
    $\text{logits} = \text{compute\_logits}(g, w) \in \mathbb{R}^N$
    **if** *second training stage* **then**
      Compute loss $\mathcal{L}_{xe}(\text{logits}, \mathbf{y})$
      Compute loss $\mathcal{L}_{PPC}(\mathbf{S}, \mathbf{A})$
      Minimize loss by updating $\mathcal{F}$
    **end if**
  **end for**
**end for**

## B. Experimental Details

**Implementations.** We train our network with a batch size of 128 and input size of 224. For experiments on all three datasets, We trained previous Prototypical-Part Networks by setting the number of part prototypes per class $K = \{3, 5\}$, and the number of added Vision Transformer blocks $m = 3$. Specifically, each identity block is copied from the 4th, 8th and 12th Transformer blocks in the original sequence, with the weights of attention projection layer and the second fully-connected layer set of zero, each inserted after the block they are copied from. We use a learning rate of $1e-4$ to train these blocks while freezing the rest of the backbone during the joint learning phase of these methods. All the other hyperparameters are the same as the ones in original implementations.

**Datasets.** CUB-200-2011 contains keypoint annotation of 15 parts and foreground segmentation masks for each image besides image-level class levels, which are used to calculate the Consistency score proposed in [19] and our Comprehensiveness score. We followed ProtoPNet [7] to perform offline data augmentation when training each method. Each image is augmented 40 times with random rotation, random skew, random shear and horizontal flip. For evaluation on Stanford Cars and Stanford Dogs, we did not perform any

| Methods | ViT-B | ViT-S |
|---|---|---|
| ProtoPNet (K=3) | 86.47 | 78.16 |
| ProtoPNet (K=5) | 86.75 | 79.51 |
| TesNet (K=3) | 87.33 | 80.97 |
| TesNet (K=5) | 87.15 | 82.55 |
| EvalProtoPNet | 91.20 | 83.40 |
| EvalProtoPNet | 91.46 | **87.09** |
| Ours (K=3) | 91.29 | 85.52 |
| Ours (K=5) | **93.63** | 86.10 |

Table 1. Classification results on Stanford Cars dataset. All methods are trained with DINOv2. Best results are in bold.

| Methods | ViT-B | ViT-S |
|---|---|---|
| ProtoPNet (K=3) | 84.11 | 79.25 |
| ProtoPNet (K=5) | 84.90 | 79.28 |
| TesNet (K=3) | 85.16 | 82.14 |
| TesNet (K=5) | 87.44 | 82.01 |
| EvalProtoPNet | 87.88 | 83.68 |
| EvalProtoPNet | 88.08 | 83.80 |
| Ours (K=3) | 87.05 | 83.11 |
| Ours (K=5) | **88.59** | **83.93** |

Table 2. Classification results on Stanford Dogs dataset. All methods are trained with DINOv2. Best results are in bold.

additional data augmentation besides resizing.

## C. Additional Results

The results on Stanford Cars and Stanford Dogs datasets over two backbones are shown in Table 1 and Table 2.

**Foreground Extraction.** We experimented with a few methods that can be directly used with our framework. PaPr [26] produces a foreground mask by thresholding the feature map from the last layer of a lightweight convolutional backbone such as resnet18 and MobileNets. Attention Rollout [46] calculates the foreground mask with the attention values between the patch tokens and the `cls` token of each Transformer block of a ViT backbone. Finally, the foreground masks can also be obtained by thresholding the first principal component of a batch of patch tokens [27]. Table 3 shows the effect of each method on the performance of our framework.

| | Con. | Sta. | Dis. | Class. |
|---|---|---|---|---|
| No foreground extraction | 29.67 | 82.85 | 94.17 | 88.42 |
| PaPr w/ resnet18 | 51.01 | 82.33 | 97.23 | 88.76 |
| Attention Rollout | 61.90 | 79.97 | 92.77 | 90.75 |
| First Principal Component | **66.40** | **82.97** | **94.45** | **90.82** |

Table 3. The effect of various off-the-shelf ViT foreground patch extraction methods on the performance of our framework.

**Algorithm 2** Python-like pseudo-code of Distinctiveness Score calculation.

```
def evaluate_distinctiveness(model, test_dataset, bbox_size):
   all_O_x = []
   for image in test_dataset:
      activations = model.get_activation(images) # shape: K, H, W
      sample_IoUs = []
      for i in range(K):
         for j in range(K):
            activation_i_max_coord = max_coord(activations[i])
            activation_j_max_coord = max_coord(activations[j])
            bbox_i = get_bbox(activation_i_max_coord)
            bbox_j = get_bbox(activation_j_max_coord)
            IoU = compute_box_IoU(bbox_i, bbox_j)
            sample_IoUs.append(IoU)
      O_x = mean(sample_IoUs)
      all_O_x.append(O_x)
   return 1 - mean(all_O_x)
```

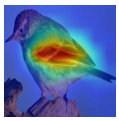
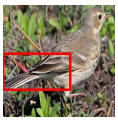
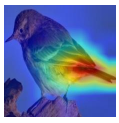
**Algorithm 3** Python-like pseudo-code of Comprehensiveness Score calculation.

```
def normalize(activations):
   activations = activations.unsqueeze(0)
   max_values = F.adaptive_max_pool2d(activations, (1, 1))
   min_values = -F.adaptive_max_pool2d(-activations, (1, 1))
   return (activations - min_values) / (max_values - min_values)

def calculate_iou(mask1, mask2, eps=1e-6):
   intersection = torch.sum(mask1 & mask2)
   union = torch.sum(mask1 | mask2)
   IoU = intersection / (union + eps)
   return IoU

def evaluate_comprehensiveness(model, test_dataset, threshold):
   all_IoUs = []
   for (image, ground_truth_foreground_map) in test_dataset:
      activations = model.get_activation(images) # shape: K, H, W
      activations = normalize(activations)
      activations_union = (activations >= threshold).sum(dim=0)
      IoU = calculate_IoU(
         activations_union,
         ground_truth_foreground_map
      )
      all_IoUs.append(IoU)
   return mean(all_IoUs)
```

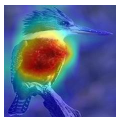
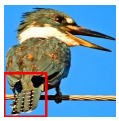
## D. Visualizations

We show the typical reasoning process of our framework across CUB-200-2011 in Table 4. Each row displays how a particular prototype contributes to the logit of its corresponding class. Note that first column displays the prototype's most similar image patch from the training set, while the second column displays the source image of the prototype in the first column. In addition, we also provide more comparison of activation maps across ProtoPNet architectures in Figure 1.

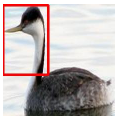
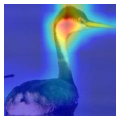
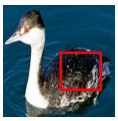
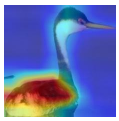
**Why is this image classified as a American Pipit ?**

| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 2.92 |
|  |  |  | 2.33 |
|  |  |  | 2.71 |
| ... | ... | ... | ... |

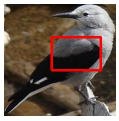
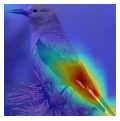
**Why is this image classified as a Ringed Kingfisher ?**

| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 3.05 |
|  |  |  | 2.43 |
|  |  |  | 2.73 |
| ... | ... | ... | ... |

**Why is this image classified as a Western Grebe ?**

| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 3.09 |
|  |  |  | 2.82 |
|  |  |  | 2.52 |
| ... | ... | ... | ... |

**Why is this image classified as a Clark Nutcracker ?**

| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 2.93 |
|  |  |  | 2.99 |
|  |  |  | 2.82 |
| ... | ... | ... | ... |

**Why is this image classified as a Great Grey Shrike ?**



| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 2.99 |
|  |  |  | 3.02 |
|  |  |  | 3.28 |
| ... | ... | ... | ... |

**Why is this image classified as a Yellow Warbler ?**



| Prototypical Part | Source Image | Activation Map | Score |
|---|---|---|---|
|  |  |  | 2.88 |
|  |  |  | 2.73 |
|  |  |  | 3.26 |
| ... | ... | ... | ... |

Table 4. Visualization of the typical visual reasoning process of our framework on CUB-200-2011 dataset.

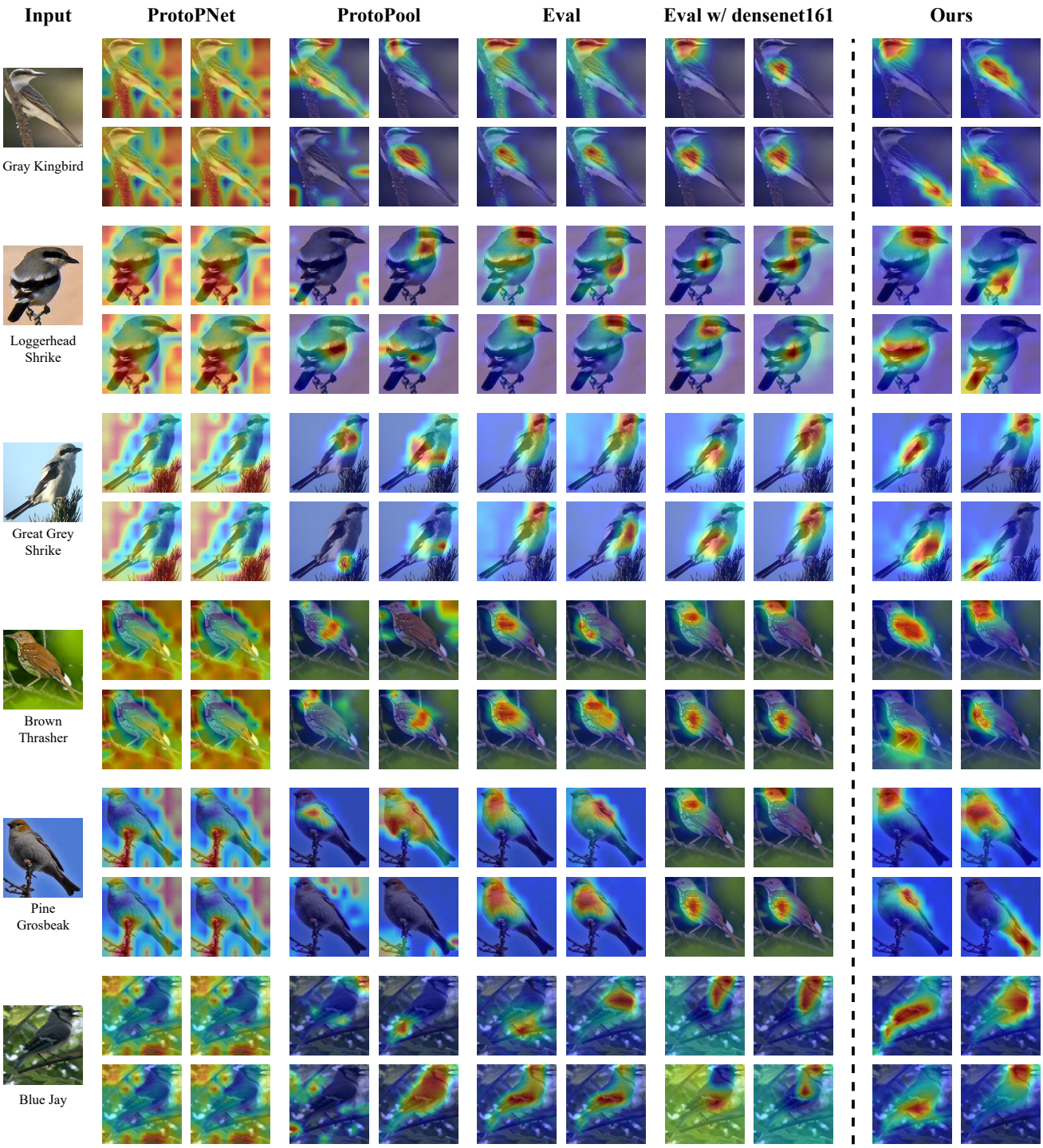



Figure 1. Visualization on CUB-200-2011 test images across various ProtoPNet architectures. All models are trained using the same DINOv2 ViT-B backbone with $K = 5$ unless annotated.