**Supplementary material for PIP-Net: Patch-Based Intuitive Prototypes for Interpretable Image Classification**

Meike Nauta  
University of Twente, the Netherlands  
m.nauta@utwente.nl

Maurice van Keulen  
University of Twente, the Netherlands  
m.vankeulen@utwente.nl

Jörg Schlötterer  
University of Duisburg-Essen, Germany  
joerg.schloetterer@uni-due.de

Christin Seifert  
University of Duisburg-Essen, Germany  
christin.seifert@uni-due.de

1. Implementation and Training Details

1.1. Training details

We use a batchsize of 128 for pretraining the prototypes, and a batch size of 64 during the second training phase with classification loss, such that PIP-Net fits on one GPU. Each mini-batch contains two views of an image, resulting in 128 inputs forwarded through f in a single mini-batch. We pretrain the prototypes of PIP-Net for 10 epochs, followed by 60 epochs for the second training phase (classification phase).

Weights $\omega_c$ of the linear layer are initialized by sampling from $\mathcal{N}(1.0, 0.1)$. Weights $< 1e^-3$ in this layer are clamped to zero, in order to prevent negative weights. During pretraining, weight $\lambda_T = 1$ for the tanh-loss is 5, and $\lambda_A$ is slowly increased to 1 (as a warm start, to prevent the trivial solution that every patch gets the same encoding). During the second training phase, weights for the losses are set to $\lambda_C = \lambda_T = 2$, $\lambda_A = 5$. We didn’t do an extensive hyperparameter search in order to save energy and computing resources, and rather chose reasonable values. For the first three epochs in the second stage, backbone $f$ is frozen and only weights $\omega_c$ of the linear layer are trained. During inference, prototype presence scores $p < 0.1$ are ignored.

1.2. Fine-grained Localization

Overlaying the $7\times7$ latent output from ResNet as used in e.g. ProtoTree [6] or ProtoPNet [1] over a $224 \times 224$ image yields non-overlapping patches of exactly $32 \times 32$ pixels. Similarly, we also use an image patch size of $32 \times 32$. For visualization purposes, the image patch can then be resized to the original image size. It is a design choice whether a prototype is visualized based on a single latent patch, as done in e.g. PIP-Net and ProtoTree, or as a larger image area containing all latent patches that are similar to the prototype as done in e.g. ProtoPNet.

Although we follow existing part-prototype methods by using an image patch size of $32 \times 32$ for an input of $224 \times 224$, we allow a more fine-grained localization by adapting the strides of ConvNeXt-tiny [3] and ResNet50. This small change keeps the model architecture compatible with pretrained weights of the standard architectures, while out-

---

**Algorithm 1: Training a PIP-Net**

```plaintext
Input: Training set $\mathcal{T}$, nEpochs
1. initialize PIP-Net with pretrained backbone $f$ with $\omega_f$ and a fully-connected layer with $\omega_c \in \mathcal{N}(1.0, 0.1)$;
2. for $t \in \{1, \ldots, nEpochs\}$ do
   3. randomly split $\mathcal{T}$ into $B$ mini-batches;
   4. for $(x_b, y_b) \in \{\mathcal{T}_1, \ldots, \mathcal{T}_b, \ldots, \mathcal{T}_B\}$ do
      /* Data Augmentation */
      5. $x_b = \text{AugWithLocation}(x_b)$;
      6. $x'_b = \text{AugNoLocation}(x_b)$;
      7. $x''_b = \text{AugNoLocation}(x_b)$;
      /* Training Prototypes */
      8. $z' = f(x'_b), z'' = f(x''_b)$;
      9. $p' = \text{MaxPool}(z'), p'' = \text{MaxPool}(z'')$;
     10. compute loss $L_A(z', z'')$;
     11. compute loss ($L_T(p') + L_T(p'')$)/2;
     12. if pretraining phase then
         13. Minimize losses by updating $\omega_f$;
        else
         14. /* Training Classifier */
            15. $p = \text{concat}(p', p'')$;
            16. $o = \log((p\omega_c)^m + 1)$;
            17. compute loss $L_C(o, y)$;
            18. Minimize losses by updating $\omega_f, \omega_c, m$;
```
putting a more fine-grained patch grid $z$. We reduced the stride of the last two layers from 2 to 1, resulting in a $28 \times 28$ output for ResNet50 and $26 \times 26$ for ConvNeXt-tiny. As a result, the latent grid still corresponds to image patches of $32 \times 32$, but then with overlap between the patches. We used CNN backbones pretrained on ImageNet, and pretrained on iNaturalist2017 \cite{7} for CUB, similar to ProtoTree \cite{6}.

### 1.3. Data Augmentation

TrivialAugment \cite{5} is used to augment the images. This recent augmentation strategy is parameter-free and only applies a single augmentation to each image. We however apply TrivialAugment twice, as shown in Fig. 1 and indicated in Line 5-7 of Algorithm 1. The first augmentation operation applies location-related transformations, including shearing, rotation and translation. This augmented image is then used as input to another TrivialAugment operation, which applies color-related transformations, including brightness, sharpness, hue and contrast. Since these transformations are randomly applied, two runs of the same function result in two different views: $x'$ and $x''$. To include even more variation between the two views, we also include a random crop of size 95% to 100% of the input. In that way, slight location variation is incorporated between the two views but an image patch at a particular location still roughly corresponds to the same pixels in both views, and can therefore be optimized to get a similar prototype.

### 1.4. CUB Prototype Purity

The CUB-200-2011 dataset \cite{8} contains pixel locations of the center of 15 different object parts. For three object parts, the dataset distinguishes between left and right (e.g., left eye and right eye). Since this difference is not needed for evaluating interpretability, we ignore the left and right specifications and select the pixel location (either left or right) that is nearest to the particular patch.

### 2. Sparsity and Compact Explanations

Sparse weights between prototypes and classes improve interpretability, as it reduces the number of relevant prototypes for a class and hence decreases explanation size. Rather than developing our own activation function, one may wonder why we don’t apply an existing sparsity method. Existing sparsity and pruning methods are mainly developed for reducing memory and computation costs \cite{2} and are therefore not directly relevant to our interpretability goal.

Specifically, often the sparsity ratio has to be predetermined by the user \cite{2, 4}, whereas we aim for maximum classification performance where the feasible sparsity ratio will depend on the dataset and classification task. Additionally, other existing sparsity methods apply pruning after training a dense model \cite{2, 4}. Instead, we can keep a dense model and are only interested in sparsifying the last linear layer. Freezing backbone $f$ and only pruning connections would imply that our prototypes are frozen rather than being optimized for sparse classification. We therefore did not apply existing sparsity methods but rather found that constructing a novel training mechanism (output scores $o$ with $L_C$) that optimizes classification performance and sparsity simultaneously was most effective.

Figure 2 shows the sparsity ratio of PIP-Net after each epoch, defined as the fraction of weights in the linear classification layer with a value $< 1e - 3$ (which are set to zero). It can be seen that PIP-Net already learns sparse connections within a few epochs, because of the pretrained prototypes. The sparsity ratio for PIP-Net-R is usually slightly higher than for PIP-Net-C since PIP-Net starts with 2048 prototypes for ResNet (i.e., 2048 * 200 weights), in contrast to 768 prototypes for ConvNeXt. We found that the last epochs mainly improve the prototype purity and have less influence on the sparsity.
3. Handling OoD Data

Fig. 3 shows more predictions of PIP-Net trained on PETS, confirming that PIP-Net behaves intuitively by design: it gives near-zero scores to out-of-distribution (OoD) data and can also handle multi-object images.

4. Prototype Visualizations

A prototype in PIP-Net is a node in the neural network that is activated (output near 1) when the prototypical part is detected in the input image, and near 0 otherwise. To visualize what a particular prototype represents, we visualize the image patches that give the highest prototype presence scores. Each row in a subfigure of Fig. 4 shows representative image patches of one prototype. It can be seen that image patches corresponding to a prototypical part look highly similar.
Figure 3. Top-3 predictions of PIP-Net trained on PETS (37 cat and dog species) for different types of input. PIP-Net can classify an image as belonging to multiple classes (middle row) or no class (bottom row). It is therefore able to deal with out-of-distribution data, making the model more intuitive for the user.

(a) Top-3 predictions of PIP-Net for in-distribution images, taken from the PETS test set. Center image shows a misclassification.

(b) Top-3 predictions of PIP-Net for multi-object images. An animal from a different class from the PETS dataset is manually pasted into an original image. PIP-Net can classify multiple objects even though it was trained on single-object images only.

(c) Top-3 predictions of PIP-Net trained on PETS for out-of-distribution images. Predictions are zero for non-animal images. PIP-Net assigns low scores for near-distribution images with animals, which is reasonable. E.g., the right image of a water buffalo gets low scores assigned since it has some (but not much) similarity with a black dog species.
Figure 4. Prototypes learned by PIP-Net (ConvNeXt backbone), one per row, visualized with their top-10 image patches. First 50 prototypes with a class-relevance weight > 0 per dataset. Some prototypes can be rare, having less than 10 similar image patches (especially for datasets such as CUB with a low number of images per class). As a result, image patches with a lower similarity to the prototype are shown to complete the top-10.
References


