CONCEPT-LEVEL DEBUGGING OF PART-PROTOTYPE NETWORKS

Andrea Bontempelli1, Stefano Teso2,1, Katya Tentori2,3, Fausto Giunchiglia1, Andrea Passerini1
1 DISI 2 CIMeC 3 DIPSOC
University of Trento
name.surname@unitn.it

ABSTRACT

Part-prototype Networks (ProtoPNets) are concept-based classifiers designed to achieve the same performance as black-box models without compromising transparency. ProtoPNets compute predictions based on similarity to class-specific part-prototypes learned to recognize parts of training examples, making it easy to faithfully determine what examples are responsible for any target prediction and why. However, like other models, they are prone to picking up confounders and shortcuts from the data, thus suffering from compromised prediction accuracy and limited generalization. We propose ProtoPDebug, an effective concept-level debugger for ProtoPNets in which a human supervisor, guided by the model’s explanations, supplies feedback in the form of what part-prototypes must be forgotten or kept, and the model is fine-tuned to align with this supervision. Our experimental evaluation shows that ProtoPDebug outperforms state-of-the-art debuggers for a fraction of the annotation cost. An online experiment with laypeople confirms the simplicity of the feedback requested to the users and the effectiveness of the collected feedback for learning confounder-free part-prototypes. ProtoPDebug is a promising tool for trustworthy interactive learning in critical applications, as suggested by a preliminary evaluation on a medical decision making task.

1 INTRODUCTION

Part-Prototype Networks (ProtoPNets) are “gray-box” image classifiers that combine the transparency of case-based reasoning with the flexibility of black-box neural networks (Chen et al., 2019). They compute predictions by matching the input image with a set of learned part-prototypes – i.e., prototypes capturing task-salient elements of the training images, like objects or parts thereof – and then making a decision based on the part-prototype activations only. What makes ProtoPNets appealing is that, despite performing comparably to more opaque predictors, they explain their own predictions in terms of relevant part-prototypes and of examples that these are sourced from. These explanations are – by design – more faithful than those extracted by post-hoc approaches (Dombrowski et al., 2019; Teso, 2019; Lakkaraju & Bastani, 2020; Sixt et al., 2020) and can effectively help stakeholders to simulate and anticipate the model’s reasoning (Hase & Bansal, 2020).

Despite all these advantages, ProtoPNets are prone – like regular neural networks – to picking up confounders from the training data (e.g., class-correlated watermarks), thus suffering from compromised generalization and out-of-distribution performance (Lapuschkin et al., 2019; Geirhos et al., 2020). This occurs even with well-known data sets, as we will show, and it is especially alarming as it can impact high-stakes applications like COVID-19 diagnosis (DeGrave et al., 2021) and scientific analysis (Schramowski et al., 2020).

We tackle this issue by introducing ProtoPDebug, a simple but effective interactive debugger for ProtoPNets that leverages their case-based nature. ProtoPDebug builds on three key observations: (i) In ProtoPNets, confounders – for instance, textual meta-data in X-ray lung scans (DeGrave et al., 2021) and irrelevant patches of background sky or foliage (Xiao et al., 2020) – end up appearing as part-prototypes; (ii) Sufficiently expert and motivated users can easily indicate which part-prototypes are confounded by inspecting the model’s explanations; (iii) Concept-level feedback of this kind is context-independent, and as such it generalizes across instances.
In short, ProtoPDebug leverages the explanations naturally output by ProtoPNets to acquire concept-level feedback about confounded (and optionally high-quality) part-prototypes, as illustrated in Fig. 1. Then, it aligns the model using a novel pair of losses that penalize part-prototypes for behaving similarly to confounded concepts, while encouraging the model to remember high-quality concepts, if any. ProtoPDebug is ideally suited for human-in-the-loop explanation-based debugging (Kulesza et al., 2015; Teso & Kersting, 2019), and achieves substantial savings in terms of annotation cost compared to alternatives based on input-level feedback (Barnett et al., 2021). In fact, in contrast to the per-pixel relevance masks used by other debugging strategies (Ross et al., 2017; Teso & Kersting, 2019; Plumb et al., 2020; Barnett et al., 2021), concept-level feedback automatically generalizes across instances, thus speeding up convergence and preventing relapse. Our experiments show that ProtoPDebug is effective at correcting existing bugs and at preventing new ones on both synthetic and real-world data, and that it needs less corrective supervision to do so than state-of-the-art alternatives.

**Contributions.** Summarizing, we: (1) Highlight limitations of existing debuggers for black-box models and ProtoPNets; (2) Introduce ProtoPDebug, a simple but effective strategy for debugging ProtoPNets that drives the model away from using confounded concepts and prevents forgetting well-behaved concepts; (3) Present an extensive empirical evaluation showcasing the potential of ProtoPDebug on both synthetic and real-world data sets.

## 2 PART-PROTOTYPE NETWORKS

ProtoPNets (Chen et al., 2019) classify images into one of \( v \) classes using a three-stage process comprising an embedding stage, a part-prototype stage, and an aggregation stage; see Fig. 1 (left).

**Embedding stage:** Let \( x \) be an image of shape \( w \times h \times d \), where \( d \) is the number of channels. The embedding stage passes \( x \) through a sequence of (usually pre-trained) convolutional and pooling layers with parameters \( \phi \), obtaining a latent representation \( z = h(x) \) of shape \( w' \times h' \times d' \), where \( w' < w \) and \( h' < h \). Let \( Q(z) \) be the set of \( 1 \times 1 \times d' \) subtensors of \( z \). Each such subtensor \( q \in Q(z) \) encodes a filter in latent space and maps a rectangular region of the input image \( x \).

**Part-prototype stage:** This stage memorizes and uses \( k \) part-prototypes \( P = \{ p_1, \ldots, p_k \} \). Each \( p_j \) is a tensor of shape \( 1 \times 1 \times d' \) explicitly learned — as explained below — so as to capture salient visual concepts appearing in the training images, like heads or wings. The activation of a part-prototype \( p \) on a part \( q \in Q(z) \) is given by a difference-of-logarithms function, defined as (Chen et al., 2019):

\[
\text{act}(p, q) := \log(\|p - q\|^2 + 1) - \log(\|p - q\|^2 + \epsilon) \geq 0 \quad (1)
\]
Here, $\|\cdot\|$ indicates the $L_2$ norm and $\epsilon > 0$ is a small constant. Alternatively, one can employ an exponential activation, defined as \cite{Hase&Ban19}:

$$act(p, q) = \exp(-\gamma \|p - q\|^2)$$  \hspace{1cm} (2)

Both activation functions are bell-shaped and decrease monotonically with the distance between $q$ and $p$, as illustrated in Fig. 2. The image-wide activation of $p$ is obtained via max-pooling, i.e.,

$$\text{act}(p, z) := \max_{q \in \mathcal{Q}(z)} \text{act}(p, q).$$

Given an input image $x$, this stage outputs an activation vector that captures how much each part-prototype activates on the image:

$$a(x) := (act(p, h(x)) : p \in \mathcal{P}) \in \mathbb{R}^k$$  \hspace{1cm} (3)

The activation vector essentially encodes the image according to the concept vocabulary. Each class $y \in [v] := \{1, \ldots, v\}$ is assigned $\lfloor \frac{v}{y} \rfloor$ discriminative part-prototypes $\mathcal{P}^y \subseteq \mathcal{P}$ learned so as to strongly activate (only) on training examples of that class, as described below.

**Aggregation stage:** In this stage, the network computes the score of class $y$ by aggregating the concept activations using a dense layer, i.e., $\sum_j w^y_j a_j(x)$, where $w^y \in \mathbb{R}^k$ is the weight vector of class $y$, and applies a softmax function to obtain conditional probabilities $p_\theta(y \mid x)$. The set of parameters appearing in the ProtoPNet will be denoted by $\theta$.

**Loss and training.** The network is fit by minimizing a compound empirical loss over a data set $D$:

$$\ell(\theta) := -\frac{1}{|D|} \sum_{(x,y) \in D} \log p_\theta(y \mid x) + \lambda_{\text{cls}}(\theta) + \lambda_{\text{sep}}(\theta)$$  \hspace{1cm} (4)

This comprises a cross-entropy loss (first term) and two regularization terms that encourage the part-prototypes to cluster the training set in a discriminative manner:

$$\ell_{\text{cls}}(\theta) := \frac{1}{|D|} \sum_{(x,y) \in D} \min_{p \in \mathcal{P}^y} \|p - q\|^2$$  \hspace{1cm} (5)

$$\ell_{\text{sep}}(\theta) := -\frac{1}{|D|} \sum_{(x,y) \in D} \min_{p \in \mathcal{P} \setminus \mathcal{P}^y} \|p - q\|^2$$  \hspace{1cm} (6)

Specifically, the part-prototypes are driven by the clustering loss $\ell_{\text{cls}}$ to cover all examples of their associated class and by the separation loss $\ell_{\text{sep}}$ not to activate on examples of other classes.

During training, the embedding and part-prototype layers are fit jointly, so as to encourage the data to be embedded in a way that facilitates clustering. At this time, the aggregation weights are fixed: each weight $w^y_j$ of $p_j \in \mathcal{P}^y$ is set to 1 if $u = y$ and to $-0.5$ otherwise. The aggregation layer is fit in a second step by solving a logistic regression problem. ProtoPNet guarantees the part-prototypes to map to concrete cases by periodically projecting them onto the data. Specifically, each $p \in \mathcal{P}^y$ is replaced with the (embedding of the closest image part from any training example of class $y$).

**Explanations.** The architecture of ProtoPNet makes it straightforward to extract explanations highlighting, for each part-prototype $p$: (1) its relevance for the target decision $(x, y)$, given by the score $w_j a_j(x)$; (2) its attribution map $\text{attr}(p, x)$, obtained by measuring the activation of $p$ on each part $q$ of $x$, and then upscaling the resulting matrix to $w \times h$ using bilinear filtering:

$$\text{attr}(p, x) := \text{upscale}(\{\text{act}(p, q)\}_{q \in \mathcal{Q}(x)});$$  \hspace{1cm} (7)

(3) the source example that it projects onto.

### 3 Input-level Debugging Strategies and Their Limitations

Explanations excel at exposing confounders picked up by models from data \cite{Lapushkin20,Geirhos20}, hence constraining or supervising them can effectively disassociate the model from acquiring those confounders. This observation lies at the heart of recent approaches for debugging machine learning models, see \cite{Teso22} for an overview.

**A general recipe.** The general strategy is well illustrated by the right for right reasons (RRR) loss \cite{Ross17}, which aligns the attribution maps produced by a predictor to those supplied by a human annotator. Let $f : x \mapsto y$ be a differentiable classifier and $IG(f, (x, y)) \in \mathbb{R}^d$ be the input gradient of $f$ for a decision $(x, y)$. This attribution algorithm \cite{Baehrens10,Simonyan14} assigns relevance to each $x_j$ for the given decision based on the magnitude
of the gradient w.r.t. $x_j$ of the predicted probability of class $y$. The RRR loss penalizes $f$ for associating non-zero relevance to known irrelevant input variables:

$$\ell_{rrr}(\theta) = \frac{1}{|D|} \sum_{(x,m,y) \in D} \| (1 - m) \odot IG(f, (x,y)) \|^2$$

(8)

Here, $m$ is the ground-truth attribution mask of example $(x, y)$, i.e., $m_j$ is 0 if the $j$-th pixel of $x$ is irrelevant for predicting the ground-truth label and 1 otherwise. Other recent approaches follow the same recipe (Rieger et al., 2020; Selvaraju et al., 2019; Shao et al., 2021; Viviano et al., 2021); see Lertvittayakumjorn & Toni (2021) and Friedrich et al. (2022) for an in-depth overview.

Debugging ProtoPNets. A natural strategy to debug a ProtoPNet with a confounded part-prototype $p$ is to delete the latter and fine-tune the remaining part-prototypes $P' := P \setminus \{p\}$. This solution is however very fragile, as nothing prevents the updated network from re-acquiring the same confounder using one of its remaining part-prototypes. One alternative is to freeze the part-prototypes $P'$ and fine-tune the aggregation stage only, but this also problematic: in the presence of a strong confound, the model ends up relying chiefly on $p$, with the remaining part-prototypes $P'$ being low quality, not discriminative, and associated to low weights. Fine-tuning the aggregation layer on $P'$ then typically yields poor prediction performance.

These issues prompted the development of the only existing debugger for ProtoPNets, IAIA-BL (Barnett et al., 2021). It also fits the above template, in that – rather than removing confounded part-prototypes – it penalizes those part-prototypes that activate on pixels annotated as irrelevant. The IAIA-BL loss is defined as:

$$\ell_{iaia}(\theta) = \frac{1}{|D|} \sum_{(x,m,y) \in D} \left( \sum_{p \in P} \| (1 - m) \odot \text{attr}(p, x) \| + \sum_{p \not\in P} \| \text{attr}(p, x) \| \right)$$

(9)

where $\text{attr}$ is the ProtoPNet’s attribution map (see Eq. (7)) and $\odot$ indicates the element-wise product. The first inner summation is analogous to the RRR loss, while the second one encourages the part-prototypes of other classes not to activate at all, thus reinforcing the separation loss in Eq. (6).

Limitations. A critical issue with these approaches is that they are restricted to pixel-level supervision, which is inherently local: an attribution map that distinguishes between object and background in a given image does not generally carry over to other images. This entails that, in order to see any benefits for more complex neural nets, a substantial number of examples must be annotated individually, as shown by our experiments (see Fig. 3). This is especially troublesome as ground-truth attribution maps are not cheap to acquire, partly explaining why most data sets in the wild do not come with such annotations.

Put together, these issues make debugging with these approaches cumbersome, especially in human-in-the-loop applications where, in every debugging round, the annotator has to supply a non-negligible number of pixel-level annotations.

4 CONCEPT-LEVEL DEBUGGING WITH PROTOPDEBUG

Our key observation is that in ProtoPNets confounders only influence the output if they are recognized by the part-prototypes, and that therefore they can be corrected for by leveraging concept-level supervision, which brings a number of benefits. We illustrate the basic intuition with an example:

Example: Consider the confounded bird classification task in Fig. 1 (right). Here, all training images of class Crested Auklet have been marked with a blue square "■", luring the network into relying heavily on this confound, as shown by the misclassified Laysian Albatro image. By inspecting the learned part-prototypes – together with source images that they match strongly, for context – a sufficiently expert annotator can easily indicate the ones that activate on confounders, thus naturally providing concept-level feedback.

Concept-level supervision sports several advantages over pixel-level annotations:

1. It is cheap to collect from human annotators, using an appropriate UI, by showing part-prototype activations on selected images and acquiring click-based feedback.
2. It is very informative, as it distinguishes between content and context and thus generalizes across instances. In our example, the blue square is a nuisance regardless of what image it appears in. In this sense, one concept-level annotation is equivalent to several input-level annotations.
3. By generalizing beyond individual images, it speeds up convergence and prevents relapse.

The debugging loop. Building on these observations, we develop ProtoPDebug, an effective and annotation efficient debugger for ProtoPNets that directly leverages concept-level supervision, which we describe next. The pseudo-code for ProtoPDebug is listed in Algorithm 1.

ProtoPDebug takes a ProtoPNet $f$ and its training set $D$ and runs for a variable number of debugging rounds. In each round, it iterates over all learned part-prototypes $p \in \mathcal{P}$, and for each of them retrieves the $a$ training examples $(x, y)$ that it activates the most on, including its source example. It then asks the user to judge $p$ by inspecting its attribution map on each selected example. If the user indicates that a particular activation of $p$ looks confounded, ProtoPDebug extracts a “cut-out” $x_R$ of the confounder from the source image $x$, defined as the box (or boxes, in case of disconnected activation areas) containing $95\%$ of the part-prototype activation. Then it embeds $x_R$ using $h$ and adds it to a set of forbidden concepts $\mathcal{F}$. This set is organized into class-specific subsets $\mathcal{F}_y$, and the user can specify whether $x_R$ should be added to the forbidden concepts for a specific class $y$ or to all of them (class-agnostic confounder).

Conversely, if $p$ looks particularly high-quality to the user, ProtoPDebug embeds its cut-out and adds the result to a set of valid concepts $\mathcal{V}$. This is especially important when confounders are ubiquitous, making it hard for the ProtoPNet to identify (and remember) non-confounded prototypes (see COVID experiments in Section 5.2).

Once all part-prototypes have been inspected, $f$ is updated so as to steer it away from the forbidden concepts in $\mathcal{F}$ while alleviating forgetting of the valid concepts in $\mathcal{V}$ using the procedure described below. Then, another debugging round begins. The procedure terminates when no confounder is identified by the user.

Fine-tuning the network. During fine-tuning, we search for updated parameters $\theta' = \{\phi', \mathcal{P}', w'\}$ that are as close as possible to $\theta$ – thus retaining all useful information that $f$ has extracted from the data – while avoiding the bugs indicated by the annotator. This can be formalized as a constrained distillation problem (Gou et al., 2021):

$$\arg\min_{\theta'} d(\theta', \theta)^2 \quad \text{s.t.} \quad \theta' \text{ is consistent with } \mathcal{F}$$

(10)

where $d$ is an appropriate distance function between sets of parameters and by “consistent” we mean that the ProtoPNet parameterized by $\theta'$ activates little or at all on the part-prototypes in $f$. Since the order of part-prototypes is irrelevant, we define it as a permutation-invariant Euclidean distance:

$$d(\theta, \theta')^2 = ||\phi - \phi'||^2 + \min_{\pi} \sum_{j \in [k]} ||p_j - p'_{\pi(j)}||^2 + ||w - w'||^2$$

(11)

Here, $\pi$ simply reorders the part-prototypes of the buggy and updated models so as to maximize their alignment. Recall that we are interested in correcting the concepts, so we focus on the middle term. Notice that the logarithmic and exponential activation functions in Eqs. (1) and (2) are both inversely proportional to $||p - p'||^2$ and achieve their maximum when the distance is zero, hence minimizing the distance is analogous to maximizing the activation. This motivates us to introduce two new penalty terms:

$$\ell_{\text{for}}(\theta) := \frac{1}{v} \sum_{y \in [v]} \max_{p \in \mathcal{P}_y} \text{act}(p, f)$$

(12)

$$\ell_{\text{rem}}(\theta) := -\frac{1}{v} \sum_{y \in [v]} \min_{p \in \mathcal{P}_y} \text{act}(p, v)$$

(13)

The hard constraint in Eq. (10) complicates optimization, so we replace it with a smoother penalty function, obtaining a relaxed formulation $\arg\min_{\theta'} d(\theta, \theta')^2 + \lambda \cdot \ell_{\text{for}}(\theta)$. The forgetting loss $\ell_{\text{for}}$ minimizes how much the part-prototypes of each class $y \in [v]$ activate on the most activated concept.
to be forgotten for that class, written $F_{u}$. Conversely, the remembering loss $\ell_{rem}$ maximizes how much the part-prototypes activate on the least activated concept to be remembered for that class, denoted $V_{u}$. The overall loss used by ProtoPDebug for fine-tuning the model is then a weighted combination of the ProtoPNet loss in Eq. (4) and the two new losses, namely $\ell(\theta) + \lambda_{r}\ell_{for}(\theta) + \lambda_{r}\ell_{rem}(\theta)$, where $\lambda_{r}$ and $\lambda_{f}$ are hyper-parameters. We analyze the relationship between Eqs. (12) and (13) and other losses used in the literature in Appendix A.

**Benefits and limitations.** The key feature of ProtoPDebug is that it leverages concept-level supervision, which sports improved generalization across instances, cutting annotation costs and facilitating interactive debugging. ProtoPDebug naturally accommodates concept-level feedback that applies to specific classes. E.g., snow being useful for some classes (say, winter) but irrelevant for others (like dog or wolf) (Ribeiro et al., 2016). However, ProtoPDebug makes it easy to penalize part-prototypes of all classes for activating on class-agnostic confounders like, e.g., image artifacts. ProtoPDebug’s two losses bring additional, non-obvious benefits. The remembering loss helps to prevent catastrophic forgetting during sequential debugging sessions, while the forgetting loss prevents the model from re-learning the same confounder in the future.

One source of concern is that, if the concepts acquired by the model are not understandable, debugging may become challenging. ProtoP Nets already address this issue through the projection step, but following Koh et al. (2020); Marconato et al. (2022), one could guide the network toward learning a set of desirable concepts by supplying additional concept-level supervision and appropriate regularization terms. More generally, like other explanation-based methods, in sensitive prediction tasks ProtoPDebug poses the risk of leaking private information. Moreover, malicious annotators may supply adversarial supervision, corrupting the model. These issues can be avoided by restricting access to ProtoPDebug to trusted annotators only.

## 5 Empirical Analysis

In this section, we report results showing how concept-level supervision enables ProtoPDebug to debug ProtoP Nets better than the state-of-the-art debugger IAIA-BL and using less and cheaper supervision, and that it is effective in both synthetic and real-world debugging tasks. All experiments were implemented in Python 3 using Pytorch (Paszke et al., 2019) and run on a machine with two Quadro RTX 5000 GPUs. Each run requires in-between 5 and 30 minutes. The full experimental setup is published on GitHub at [https://github.com/abonte/protopdebug](https://github.com/abonte/protopdebug) and in the Supplementary Material together with additional implementation details. ProtoPDebug and IAIA-BL were implemented on top of ProtoP Nets (Chen et al., 2019) using the Adam optimizer (Kingma & Ba, 2014). We used the source code of the respective authors of the two methods (cod, 2019; 2021b).

### 5.1 Concept-Level vs Instance-Level Debugging

We first compare the effectiveness of concept-level and instance-level supervision in a controlled scenario where the role of the confounders is substantial and precisely defined at design stage. To this end, we modified the CUB200 data set (Wah et al., 2011), which has been extensively used for evaluating ProtoP Nets (Chen et al., 2019; Hase & Bansal, 2020; Rymarczyk et al., 2021; Hoffmann...
et al., 2021; Nauta et al., 2021). This data set contains images of 200 bird species in natural environments, with approximately 40 train and 20 test images per class. We selected the first five classes, and artificially injected simple confounders in the training images for three of them. The confounder is a colored square of fixed size, and it is placed at a random position. The color of the square is different for different classes, but it is absent from the test set, and thus acts as a perfect confounder for these three classes. The resulting synthetic dataset is denoted CUB5box.

Competitors and implementation details. We compare ProtoPDebug, with supervision on a single instance per confounder, against the following competitors: 1) vanilla ProtoPNets; 2) a vanilla ProtoPNet trained on a clean version of the dataset without the confounders, denoted ProtoPNets_clean; 3) IAIA-BL X, the IAIA-BL model fit using ground-truth attribution maps on X% of the training examples, for X ∈ {5, 20, 100}, and IAIA-BL n = 3, with one attribution mask per confound (same number given to ProtoPDebug). Note that this setting is not interactive, but supervision is made available at the beginning of training. The embedding layers were implemented using a pre-trained VGG-16, allocating two prototypes for each class. The values of λ and α were set to 100 and 5, respectively, and λiaia to 0.001, as in the original paper (Barnett et al., 2021); experiments with different values of λiaia gave inconsistent results. In this setting, all supervision is made available to all methods from the beginning, and the corrective feedback fed to the forgetting loss is class-specific.

Results. Fig. 3 reports the experimental results on all methods averaged over 15 runs. The left and middle plots show the training set macro F1 and cross entropy, respectively. After 15 epochs, all methods manage to achieve close to perfect F1 and similar cross entropy. The right plot shows macro F1 on the test set. As expected, the impact of the confounders is rather disruptive, as can been seen from the difference between ProtoPNets and ProtoPNets_clean. Indeed, ProtoPNets end up learning confounders whenever present. Instance-level debugging seems unable to completely fix the problem. With the same amount of supervision as ProtoPDebug, IAIA-BL n = 3 does not manage to improve over the ProtoPNets baseline. Increasing the amount of supervision does improve its performance, but even with supervision on all examples IAIA-BL still fails to match the confound-free baseline ProtoPNets_clean (IAIA-BL 100 curve). On the other hand, ProtoPDebug succeeds in avoiding to be fooled by confounders, reaching the performance of ProtoPNets_clean despite only receiving a fraction of the supervision of the IAIA-BL alternatives. Note however that ProtoPDebug does occasionally select a natural confounder (the water), that is not listed in the set of forbidden ones. In the following section we show how to deal with unknown confounders via the full interactive process of ProtoPDebug.

5.2 ProtoPDebug in the real world

CUB5nat data set. Next, we evaluate ProtoPDebug in a real world setting in which confounders occur naturally in the data as, e.g., background patches of sky or sea, and emerge at different stages of training, possibly as a consequence of previous user corrections. To maximize the impact of the natural confounders, we selected the 20 CUB200 classes with largest test F1 difference when training ProtoPNets only on birds (removing background) or on entire images (i.e., bird + background), and testing them on images where the background has been shuffled among classes. Then, we applied ProtoPDebug focusing on the five most confounded classes out of these 20. We call this dataset CUB5nat. All architectures and hyperparameters are as before.

Experimental Protocol. To evaluate the usability and effectiveness of ProtoPDebug with real users, we ran an online experiment, through Prolific Academic (prolific.com), one of the most reliable crowdsourcing platforms for behavioral research. Participants were laypeople, all native speakers of English. To make the experimental stimuli as clear as possible, separate patches for the activation area of the prototype on an image were automatically extracted from the 5 most-activate images (a = 5). See Appendix H for the detailed procedure. For each debugging round, a new group of 10 participants was asked to inspect the images and to determine whether the highlighted patch covered some part of the bird or whether, instead, it covered exclusively (or very nearly so) the background. A patch was flagged as a confounder when 80% of the users indicated it as covering the background. A prototype was accepted when no confounder was identified in its most activated image. Regardless of this, confounders in less activated images of the prototype were still collected as they might serve as feedback for other prototypes (class-agnostic confounders) in possible following iteration(s). The procedure ended when all prototypes were accepted.
Figure 4: **Left:** Three rounds of sequential debugging with ProtoPDebug on CUB5$^{nat}$. **Rows:** part-prototypes (two per class) and user feedback (checkmark vs. cross). Note that the prototypes produced before the first correction (left-most column) correspond to those learned by plain ProtoPNets. **Right:** Test set performance of ProtoPNets and ProtoPDebug on CUB5$^{nat}$. $AP_i$ is the attribution precision of $i$-th part-prototype. Bold denotes better performance.

<table>
<thead>
<tr>
<th>1ST ROUND</th>
<th>2ND ROUND</th>
<th>3RD ROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
</tr>
<tr>
<td><img src="image4" alt="Image" /></td>
<td><img src="image5" alt="Image" /></td>
<td><img src="image6" alt="Image" /></td>
</tr>
<tr>
<td><img src="image7" alt="Image" /></td>
<td><img src="image8" alt="Image" /></td>
<td><img src="image9" alt="Image" /></td>
</tr>
<tr>
<td><img src="image10" alt="Image" /></td>
<td><img src="image11" alt="Image" /></td>
<td><img src="image12" alt="Image" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>$F_1$</th>
<th>$AP_1$</th>
<th>$AP_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.48</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>6</td>
<td>0.38</td>
<td>0.73</td>
<td>0.48</td>
</tr>
<tr>
<td>8</td>
<td>0.67</td>
<td>0.27</td>
<td>0.93</td>
</tr>
<tr>
<td>14</td>
<td>0.51</td>
<td>0.79</td>
<td>0.79</td>
</tr>
<tr>
<td>15</td>
<td>0.77</td>
<td>0.85</td>
<td>0.86</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>0.56</strong></td>
<td><strong>0.68</strong></td>
<td><strong>0.68</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>$F_1$</th>
<th>$AP_1$</th>
<th>$AP_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.54</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>6</td>
<td>0.64</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>8</td>
<td>0.93</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td>14</td>
<td>0.52</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>15</td>
<td>0.82</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td><strong>Avg.</strong></td>
<td><strong>0.69</strong></td>
<td><strong>0.90</strong></td>
<td><strong>0.90</strong></td>
</tr>
</tbody>
</table>

**Results.** The average evaluation time for each image was 6 seconds. Across all images, participants’ inter-annotation agreement was 96%, confirming the simplicity of the users’ feedback required by ProtoPDebug. **Fig. 4** shows the part-prototypes progressively learned by ProtoPDebug. In the first iteration, when no corrective feedback has been provided yet, the model learns confounders for most classes: the branches for the second and the last classes, the sky for the third one. Upon receiving user feedback, ProtoPDebug does manage to avoid learning the same confounders again in most cases, but it needs to face novel confounders that also correlate with the class (i.e., the sea for the first class). After two rounds, all prototypes are accepted, and the procedure ends. **Fig. 4** reports the test set performance of ProtoPNets compared to ProtoPDebug in terms of $F_1$ and interpretability metric. The latter has been introduced in (Barnett et al., 2021, Eq. 6) to quantify the portion of pixels that are activated by a prototype. We slightly modified it to consider also their activation value (see Suppl. for the definition). Results show that ProtoPDebug improves over ProtoPNets in all classes (with an increase in $F_1$ of up to 0.26 for classes 6 and 8), and that it manages to learn substantially more interpretable prototypes (i.e., it is right for the right reason (Schramowski et al., 2020)), with a 22% improvement in activation precision on average.

**COVID data set.** The potential of artificial intelligence and especially deep learning in medicine is huge (EJ., 2019). On the other hand, the medical domain is known to be badly affected by the presence of confounders (Jager et al., 2008; Smith & Nichols, 2018). In order to evaluate the potential of ProtoPDebug to help addressing this issue, we tested it on a challenging real-world problem from the medical imaging domain. The task is to recognize COVID-19 from chest radiographies. As shown in DeGrave et al. (2021), a classifier trained on this dataset heavily relies on confounders that correlate with the presence or absence of COVID. These confounders come from the image acquisition procedure or annotations on the image. We trained and tested ProtoPDebug on the same datasets used in DeGrave et al. (2021) using the data pipeline source code (cod, 2021a). To simplify the identification of confounders, we focused on a binary classification task, discriminating COVID-positive images from images without any pathology. Given the extremely high inter-annotator agreement of
Figure 5: Four rounds of sequential debugging with ProtoPDebug on COVID. Only the prototypes with non-zero activation are reported: first prototype refers to COVID- and second COVID+.

<table>
<thead>
<tr>
<th>1st Round</th>
<th>2nd Round</th>
<th>3rd Round</th>
<th>4th Round</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
<td><img src="image4" alt="Image" /></td>
</tr>
</tbody>
</table>

the CUB5\textsubscript{nat} experiment, this experiment was conducted internally in our lab. We leave a crowdsourced evaluation to an extended version of this work.

**Results.** Fig. 5 reports the (non-zero activation) prototypes of ProtoPDebug at different correction rounds. As for Fig. 4 (left), the left-most column corresponds to the prototypes learned by ProtoPNets. Note that for each prototype, the supervision is given to the 10 most-activated images ($a = 10$). Penalized confounders have been extracted from images on which the prototype has non-zero activation, because they influence the classification. However, the patches of the images to remember are extracted even if the activation is zero, in order to force the prototype to increase the activation on them thanks to the remembering loss. Eventually, ProtoPDebug manages to learn non-confounded prototypes, resulting in substantially improved test classification performance. The test $F_1$ goes from 0.26 of ProtoPNets (first column) to 0.54 at the end of the debugging process.

6 RELATED WORK

**Input-level debuggers.**  ProtoPDebug is inspired by approaches for explanatory debugging (Kulesza et al., 2015) and explanatory interactive learning (Teso & Kersting, 2019; Schramowski et al., 2020), which inject explanations into interactive learning, enabling a human-in-the-loop to identify bugs in the model's reasoning and fix them by supplying corrective feedback. They leverage input attributions (Teso & Kersting, 2019; Selvaraju et al., 2019; Lertvittayakumjorn et al., 2020; Schramowski et al., 2020), example attributions (Teso et al., 2021; Zylberajch et al., 2021), and rules (Popordanoska et al., 2020), but are not designed for concept-based models nor leverage concept-level explanations. IAIA-BL (Barnett et al., 2021) adapts these strategies to PPNNets by penalizing part-prototypes that activate on irrelevant regions of the input, but it is restricted to instance-level feedback, which is neither as cheap to obtain nor as effective as concept-level feedback, as shown by our experiments. For an overview of interactive debugging strategies, see Teso et al. (2022).

**Concept-level debuggers.**  Stammer et al. (2021) debug neuro-symbolic models by enforcing logical constraints on the concepts attended to by a network’s slot attention module (Locatello et al., 2020), but requires the concept vocabulary to be given and fixed. ProtoPDebug has no such requirement, and the idea of injecting prior knowledge (orthogonal to our contributions) could be fruitfully integrated with it. ProSeNets (Ming et al., 2019) learn (full) prototypes in the embedding space given by a sequence encoder and enable users to steer the model by adding, removing and manipulating the learned prototypes. The update step however requires input-level supervision and adapts the embedding layers only. iProto-TREX (Schramowski et al., 2021) combines ProtoPNets with transformers, and learns part-prototypes capturing task-relevant text snippets akin to rationales (Zaidan et al., 2007). It supports dynamically removing and hot-swapping bad part-prototypes, but it lacks a forgetting loss and hence is prone to relapse. Applying ProtoPDebug to iProto-TREX is straightforward and would fix these issues. Bontempelli et al. (2021) proposed a classification of debugging strategies for concept-based models, but presented no experimental evaluation. Lage & Doshi-Velez (2020) acquire concepts by eliciting concept-attribute dependency information with questions like “does the concept depression depend on feature lorazepam?”. This approach can prevent confounding, but it is restricted to white-box models and interaction with domain experts. FIND (Lertvittayakumjorn et al., 2020) offers similar functionality for deep NLP models, but it relies on disabling concepts only.

**Other concept-based models.** There exist several concept-based models (CBMs) besides ProtoPNets, including self-explainable neural networks (Alvarez-Melis & Jaakkola, 2018), concept bottle-
neck models (Koh et al., 2020; Losch et al., 2019), and concept whitening (Chen et al., 2020). Like ProtoPNets, these models stack a simulatable classifier on top of (non-prototypical) interpretable concepts. Closer to our setting, prototype classification networks (Li et al., 2018) and deep embedded prototype networks (Davoudi & Komeili, 2021) make predictions based on embedding-space prototypes of full training examples (rather than parts thereof). Our work analyzes ProtoPNets as they perform comparably to other CBMs while sporting improved interpretability (Chen et al., 2019; Hase & Bansal, 2020). Several extensions of ProtoPNets have also been proposed (Hase et al., 2019; Rymarczyk et al., 2021; Nauta et al., 2021; Kraft et al., 2021). ProtoPDebug naturally applies to these variants, and could be extended to other CBMs by adapting the source example selection and update steps. An evaluation on these extensions is left to future work.

Ethics statement  ProtoPDebug is conceived for leveraging interactive human feedback to improve the trustworthiness and reliability of machine predictions, which is especially important for critical applications like medical decision making. On the other hand, adversarial annotators might supply corrupted supervisions to alter the model and drive its predictions towards malignant goals. For these reasons, access to the model and to the explanations should be restricted to authorized annotators only, as standard in safety-critical applications. Concerning the online experiment with laypeople, all participants were at least 18 years old and provided informed consent prior to participating in the research activities.

Reproducibility statement  The full experimental setup is published on GitHub at at https://github.com/abonte/protopdebug. The repository contains the configuration files reporting the hyper-parameters of all experiments, and the procedures to process the data and reproduce the experiments. The implementation details are also available in Appendix C, while Appendix D informs about the data sets, which are all made freely available by their respective authors. For CUB200 (Wah et al., 2011) and image masks (Farrell, 2022), the image pre-processing reuses the code of the ProtoPNets authors (cod, 2019). The COVID images are processed using (cod, 2021a). The images are downloaded from GitHub-COVID repository (Cohen et al., 2020), ChestX-ray14 repository (Wang et al., 2017), PadChest (Bustos et al., 2020) and BIMCV-COVID19+ (Vayá et al., 2020). Appendix H provides the experimental protocol for the online experiment with real users.

ACKNOWLEDGMENTS

This research of FG has received funding from the European Union’s Horizon 2020 FET Proactive project “WeNet - The Internet of us”, grant agreement No. 823783, The research of AB was supported by the “DELPhi - DiscovEring Life Patterns” project funded by the MIUR Progetti di Ricerca di Rilevante Interesse Nazionale (PRIN) 2017 – DD n. 1062 del 31.05.2019. The research of ST and AP was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

REFERENCES


Ann-Kathrin Dombrowski, Maximillian Alber, Christopher Anders, Marcel Ackermann, Klaus-Robert Müller, and Pan Kessel. Explanations can be manipulated and geometry is to blame. *NeurIPS*, 2019.


Andrew Slavin Ross, Michael C Hughes, and Finale Doshi-Velez. Right for the right reasons: training differentiable models by constraining their explanations. In IJCAI, 2017.


A RELATIONSHIP TO OTHER LOSSES

The forgetting loss in Eq. (12) can be viewed as a lifted version of the separation loss in Eq. (6): the former runs over classes and leverages per-class concept feedback, while the latter runs over examples. The same analogy holds for the remembering and clustering losses, cf. Eqs. (5) and (13).

The other difference is that the instance-level losses are proportional to the negative squared distance $-\|p - q\|^2$ between part-prototypes and parts and the concept-level ones to the activation instead. However, existing activation functions are monotonically decreasing in the squared distance, meaning that they only really differ in fall-off speed, as can be seen in Fig. 2. The IAIA-BL loss can also be viewed as a (lower bound of the) separation loss. Indeed:

$$\ell_{iaia}(\theta) = \sum_{i} \sum_{p \in P^y} \| (1 - m) \odot \text{attr}(p, x) \| \leq \sqrt{k} \sum_{(x, m, y)} \max_{p \in P^y} \| (1 - m) \odot \text{attr}(p, x) \|_{\infty}$$

$$= \sqrt{k} \sum_{(x, m, y)} \max_{p \in P^y} |(1 - m_q) \cdot \text{act}(p, q)| = -\sqrt{k} \sum_{(x, m, y)} \min_{p, q} (1 - m_q) \cdot \|p - q\|^2$$

where $m_q \geq 0$ is the “part-level” attribution mask for part $q$, obtained by down-scaling the input-level attribution mask of all pixels in the receptive field of $q$. This formulation highlights similarities and differences with our forgetting loss: the latter imposes similar constraints as the IAIA-BL penalty, but (i) it does not require full mask supervision, and (ii) it can easily accommodate concept-level feedback that targets entire sets of examples, as explained above.

B ACTIVATION PRECISION

In order to measure the quality of the explanations produced by the various models, we adapted the activation precision (AP) metric from the original IAIA-BL experiments (Barnett et al., 2021, Eq. 6). Assuming that the attribution mask attr output by ProtoPNets is flattened to a vector with the same shape as $m$, the original formulation of AP can be written as:

$$\frac{1}{|D|} \sum_{(x, m, y) \in D} \left( \sum_i \frac{|m \odot T_{\tau}(\text{attr}(p, x))|}{|T_{\tau}(\text{attr}(p, x))|} \right)$$

Here, $T_{\tau}$ is a threshold function that takes an attribution map (i.e., a real-valued matrix), computes the $(100 - \tau)$ percentile of the attribution values, and sets the elements above this threshold to 1 and the rest to 0. Intuitively, the AP counts how many pixels predicted as relevant by the network are actually relevant according to the ground-truth attribution mask $m$.

This definition however ignores differences in activation across the most relevant pixels, which can be quite substantial, penalizing debugging strategies that manage to correctly shift most of the
network’s explanations over the truly relevant regions. To fix this, we modified $T_\tau$ not to discard this information, as follows:

$$
\frac{1}{|D|} \sum_{(x,m,y) \in D} \left( \frac{\sum_i [m \odot T_\tau(\text{attr}(p,x)) \odot \text{attr}(p,x)]_i}{\sum_i [T_\tau(\text{attr}(p,x)) \odot \text{attr}(p,x)]_i} \right)
$$

(15)

In our experiments, we set $\tau = 5\%$. The same value is used by IAIA-BL and ProtoPNets for visualizing the model’s explanations and is therefore what the user would see during interaction.

C IMPLEMENTATION DETAILS

**Architectures.** We use the same ProtoPNet architecture for all competitors:

- **Embedding step:** For the CUB experiments, the embedding layers were taken from a pre-trained VGG-16, as in (Chen et al., 2019), and for COVID from a VGG-19 (Simonyan & Zisserman, 2014).

- **Adaptation layers:** As in (Chen et al., 2019), two $1 \times 1$ convolutional layers follow the embedding layers, and use ReLu and sigmoid activation functions respectively. The output dimension is $7 \times 7$.

- **Part-prototype stage:** All models allocate exactly two part-prototypes per class, as allocating more did not bring any benefits. The prototypes are tensors of shape $1 \times 1 \times 128$.

To stabilize the clustering and separation losses, we implemented two “robust” variants proposed by Barnett et al. (2021), which are obtained by modifying the regular losses in Eqs. 5 and 6 of the main text to consider the average distance to the closest 3 part-prototypes, rather than the distance to the closest one only. These two changes improved the performance of all competitors.

**Training.** We froze the embedding layer so as to drive the model toward updating the part-prototypes. Similarly to the ProtoPNet implementation, the learning rate of the prototype layer was scaled by 0.15 every 4 epochs. The training batch size is set to 20 for the experiments on CUB$_{box}$ and COVID data sets, and 128 on CUB$_{nat}$.

We noticed that, in confounded tasks, projecting the part-prototypes onto the nearest latent training patch tends the prototypes closer to confounders, encouraging the model to rely on it. This strongly biases the results against the ProtoPNet baseline. To avoid this problem, in our experiments we disabled the projection step.

**Hyper-parameters.** The width of the activation function (Eq. 1 in the main paper) used in $\ell_{for}$ was set to $\epsilon = 10^{-8}$ in all experiments.

For CUB$_{box}$ and CUB$_{nat}$, the weights of the different losses were set to $\lambda_{cls} = 0.5$, $\lambda_{sep} = 0.08$, $\lambda_{for} = 100$. In this experiment, the remembering loss was not necessary and therefore it was disabled by setting $\lambda_{rem} = 0$. The value of $\lambda_{bias} = 0.001$ was selected from $\{0.001, 0.01, 1.0, 100\}$ so as to optimize the $F_1$ test set performance, averaged over three runs. The number of retrieved training examples shown to the user is 5 ($a = 5$).

For the COVID data set, $\lambda_{cls}$, $\lambda_{sep}$, $\lambda_{for}$ and $\lambda_{rem}$ were set to 0.5, 0.08, 200, and 0 respectively for the first two debugging rounds. In the last round, $\lambda_{cls}$, $\lambda_{sep}$ were decreased to 25% of the above values and $\lambda_{rem}$ increased to 0.01, to boost the effect of $\ell_{for}$ and $\ell_{rem}$. This shifts the focus of training from avoiding very bad part-prototypes, to adjusting the partly wrong prototypes and increasing the activation on the relevant patches. The value of $a$ is set to 10.

D DATA STATISTICS AND PREPARATION

Table 1 reports statistics for all data sets used in our experiments, including the number of training and test examples, training examples used for the visualization step, and classes. The visualization data set contains the original images, i.e., before the augmentation step, and is used to visualize the images on which the prototypes activate most during the debugging loop. All images were resized to $224 \times 224 \times 3$, as done in ProtoPNet (Chen et al., 2019), IAIA-BL (Barnett et al., 2021) and for the COVID-19 data in (DeGrave et al., 2021).
Table 1: Statistics for all data sets used in our experiments.

<table>
<thead>
<tr>
<th>Data Set</th>
<th># Train</th>
<th># Test</th>
<th># Visualization</th>
<th># of Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUB5\textsubscript{box}</td>
<td>4500</td>
<td>132</td>
<td>150</td>
<td>5</td>
</tr>
<tr>
<td>CUB5\textsubscript{nat}</td>
<td>180000</td>
<td>569</td>
<td>600</td>
<td>20</td>
</tr>
<tr>
<td>COVID</td>
<td>7640</td>
<td>1744</td>
<td>7640</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 6: From left to right, the images of confounders on which $\ell_{for}$ relies for computing the activation value: the confounding green box present in CUB5\textsubscript{box} and two background patches in CUB5\textsubscript{nat}.

D.1 CUB data sets

Classes in the CUB200 data set include only 30 training examples each. To improve the performance of ProtoPNets, as done in (Chen et al., 2019), we augmented the training images using random rotations, skewing, and shearing, resulting in 900 augmented examples per class.

CUB5\textsubscript{box}. The five CUB200 classes that were make up the CUB5\textsubscript{box} data set are listed Fig. 7 (left). Synthetic confounders (i.e., colored boxes) have been added to training images of the first, second and last class. No confounder was added to the other two classes. Fig. 6 (left) shows one of the three colored boxes. An example image confounded with a green box “■” is shown in Fig. 7 (right) along with the corresponding ground-truth attribution mask $m$ used for IAIA-BL.

Figure 7: Left: List of CUB200 classes selected for the CUB5\textsubscript{box} data set. Right: Confounded training images from CUB5\textsubscript{box} and corresponding ground-truth attribution mask.

| # | Class | Label              | # Confound?
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>Black-footed Albatross</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>002</td>
<td>Laysan Albatross</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>003</td>
<td>Sooty Albatross</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>004</td>
<td>Groove-billed Ani</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>005</td>
<td>Crested Auklet</td>
<td>✓</td>
</tr>
</tbody>
</table>

CUB5\textsubscript{nat}. We created a variation of CUB200, namely CUB5\textsubscript{nat}, to introduce a confounder that occurs naturally, e.g., sky or trees patches in the background, and are learned during the training phase. We followed these steps to select the twenty classes reported in Table 2 and used for the experiments:

1. We modified the test set images of all the 200 classes of CUB200 by placing the bird of one class on the background of another one. Since the outline of the bird removed from the background may not match the pasted one, the uncovered pixels are set to random values. Figure Fig. 8 shows the background swap for one image and the corresponding ground-truth mask. This background shift causes a drop in the performance of the model that relies on the background to classify a bird.

2. We trained ProtoPNet on two variations of the training set: in the first, by removing the background, the images contain only the birds, and in the second, both the bird and the background. We selected the twenty classes with the maximum gap in terms of $F_1$ on the
Figure 8: From left to right: original image with the sea in the background, the same bird but with the background of the land and the corresponding ground-truth attribution mask.

modified test set, out of the 200 classes of CUB200. Thus, the background appears to be a confounder for these twenty classes. The right most images in Fig. 6 highlight patches of the background learned by the model.

3. We repeated the step above but only on the twenty classes, and we selected the five classes with the maximum gap. In the experiments, we provided supervision only on these five classes. Since the experiments are run on this subset of twenty classes, the separation loss might force the model to learn different prototypes with respect to the ones learned when trained on 200 classes. Hence, the prototypes activate on different pixels and the classes with the largest gap are different. This additional ranking of the classes allows us to debug the most problematic ones and to show the potential performance improvement on them. For this reason, the results reported in the main paper are computed on only these five classes.

Table 2: Twenty classes with largest train-test performance gap used in the our second experiment. The debug column indicates which classes received supervision and are thus debugged.

<table>
<thead>
<tr>
<th>#</th>
<th>Class</th>
<th>Label</th>
<th>Debug?</th>
<th>#</th>
<th>Class</th>
<th>Label</th>
<th>Debug?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001 Black-footed Albatross</td>
<td></td>
<td>✓</td>
<td>10</td>
<td>085 Horned Lark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>004 Groove-billed Ani</td>
<td></td>
<td></td>
<td>11</td>
<td>086 Pacific Loon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>006 Least Auklet</td>
<td>12 113 Baird Sparrow</td>
<td></td>
<td>13</td>
<td>122 Harris Sparrow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>026 Bronzed Cowbird</td>
<td>14 128 Seaside Sparrow</td>
<td>✓</td>
<td>15</td>
<td>137 Cliff Swallow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>037 Acadian Flycatcher</td>
<td>14 128 Seaside Sparrow</td>
<td>✓</td>
<td>16</td>
<td>161 Blue-winged Warbler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>040 Olive-sided Flycatcher</td>
<td>15 137 Cliff Swallow</td>
<td>✓</td>
<td>17</td>
<td>183 Northern Waterthrush</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>043 Yellow-bellied Flycatcher</td>
<td>✓ 16 161 Blue-winged Warbler</td>
<td></td>
<td>18</td>
<td>188 Pileated Woodpecker</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>056 Pine Grosbeak</td>
<td>17 183 Northern Waterthrush</td>
<td></td>
<td>19</td>
<td>200 Common Yellowthroat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>070 Green Violetear</td>
<td>18 188 Pileated Woodpecker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>076 Dark-eyed Junco</td>
<td>19 200 Common Yellowthroat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D.2 COVID DATA SET

The training set consists of a subset of images retrieved from the GitHub-COVID repository (Cohen et al., 2020) and from the ChestX-ray14 repository (Wang et al., 2017). The number of COVID-negative and COVID-positive radiographs is 7390 and 250, respectively. The test set is the union of PadChest (Bustos et al., 2020) and BIMCV-COVID19+ (Vayá et al., 2020) datasets, totalling 1147 negative and 597 positive images. In (DeGrave et al., 2021), the classifier is trained on 15 classes, i.e., COVID and other 14 pathologies. In (DeGrave et al., 2021), the classifier is trained on a multi-label data set, in which each scan is associated with multiple pathologies, and the evaluation is then performed on COVID positive vs. other pathologies. To simplify the identification and thus the supervision given to the machine, we generated a binary classification problem containing COVID-positive images and images without any pathology. In this setting, supervision is given only on obvious confounders like areas outside the body profile or on arms and elbows. A domain expert can provide detailed supervision on which parts of the lungs the machine must look at.

E ADDITIONAL RESULTS

Additional part-prototypes for COVID. Fig. 9 reports all prototypes for the experiment on the COVID data set.
Confusion matrices. The confusion matrix on the left side of Fig. 10 shows that ProtoPNet overpredicts the COVID-positive class, whereas the prediction of ProtoPDebug after three rounds is more balanced since the confounders have been debugged. A domain expert can give additional supervision on which parts of the lungs are relevant for the classification task, further improving the performance.

Figure 9: Four rounds of sequential debugging with ProtoPDebug on COVID. Top row reports the prototypes with non-zero activation: first prototype refers to COVID- and second COVID+. Second row: prototypes with zero activation for each class.

<table>
<thead>
<tr>
<th>1ST ROUND</th>
<th>2ND ROUND</th>
<th>3RD ROUND</th>
<th>4TH ROUND</th>
</tr>
</thead>
<tbody>
<tr>
<td>COVID-</td>
<td>COVID+</td>
<td>COVID-</td>
<td>COVID+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Confusion Matrices on test set where class 0 and 1 is COVID-negative and COVID-positive respectively.

F COMPARISON TO REMOVING-AND-FINE-TUNING

One alternative strategy for debugging ProtoP Nets consists of removing confounded part-prototypes (or fixing their weight to zero) and then fine-tuning the aggregation layer. Here, we highlight some limitations of this strategy that ProtoPDebug does not suffer from. In all experiments we use the same setup as in the experiments reported in the main text.

First, this strategy cannot be used when all prototypes are confounded. This can occur because, in ProtoP Nets, there is no term encouraging the network to learn distinct part-prototypes. In practice, it can happen that all part-prototypes learn to recognize the same confound. This is illustrated by the two left images in Fig. 11: here a ProtoPNet trained on the CUB5 box data set has learned the same confounder (the colored box) with both of the part-prototypes allocated to the target class. In this situation, the above strategy cannot be applied as there are no unconfounded concepts to rely on for prediction.

What happens if only some part-prototypes are confounded? In order to answer this question, we assume that the ProtoPNet loss is augmented with a diversification term $\ell_{\text{div}}$ of the form:

$$\ell_{\text{div}}(\theta) := \frac{1}{|P|} \sum_{y} \sum_{p_j \in P_y} \min_{p_j' \in P_y} \| p_j - p_j' \|^2$$

where $\lambda_{\text{div}}$ is a hyper-parameter. This term encourages the different part-prototypes of each class $y$ to be different from each other, and is useful to prevent them to all acquire the same confound.
Figure 11: **Left pair:** prototype activations of a confounded class in CUB5box. Notice that both prototypes are confounded. **Right pair:** one prototype activates on the confounder (colored box) whereas the other is neither interpretable nor useful for prediction.

Now, even if only some part-prototypes are confounded, there is no guarantee that the unconfounded ones capture meaningful information. The opposite tends to be true: when a strong confounder is present, the network can (and typically does) achieve high accuracy on the training set by learning to recognize the confounder and allocating all weight to it, and has little incentive to learn other discriminative, high-quality part-prototypes. We validated this empirically by training a ProtoPNet model on the CUB5box data set using the diversification term above, setting $\lambda_{\text{div}}$ to 0.1. The results are shown in Fig. 11 (right): the two activation maps show that one part-prototype activates on the confound, while the other captures irrelevant information and that therefore it would perform very poorly when used for classification.

G ABLATING THE REMEMBERING LOSS

We evaluate the contribution of the remembering loss in last debugging rounds of the COVID experiment. The remembering loss avoids performance drop over time. We re-run the last two debugging rounds of the experiment in Figure 5 by setting $\lambda_r$ to zero. In terms of $F_1$, the performance in the third round goes from 0.55 with the remembering loss to 0.44 without. In the fourth round, performance decrease from 0.54 to 0.4.

H USER EXPERIMENT

We evaluated the usability of ProtoPDebug in a sequence of debugging round with real users. The participants were asked to review patches of prototype activation on the training images. The patch extraction procedure starts by selecting the five most activated images for each prototype. For each activation map, we generate a binary image in which pixels are set to one if their activation value is at least 95%-percentile of all activation (as in (Chen et al., 2019)), zero otherwise. Then, we extract the contours representing the border of the activation patches. We discard the patches that are too small, difficult to be spotted by the participants and not capturing enough information. We set the threshold to patches having less than 200 pixels area. Finally, we overlay the original images with the extracted patches.

We ran the online experiment through Prolific Academic. For each debugging round experiment, ten participants, laypeople and English native speakers, inspected the images generated as described above in a sequence of questions like Fig. 12. Each experiment took between 9 and 15 minutes according to the number of patches to review. Participants received an hourly fee for their participation. On top of this amount, we gave an additional bonus based on the agreement with the other participants. We randomly selected five images, and we awarded the users whose evaluations agree with at least 80% of others’ evaluations for all five images.

We provided the following instruction to the user:

*You will now be presented with a series of images of birds in various natural settings. Each image has a highlighted overlay that goes from dark red (the highlight focus) to light green (the highlight margin).*
Figure 12: Question in the user experiment form. The upper part display one activation patch. The user must choose either that the overlay covers part of the bird or exclusively the background.
Table 3: Statistics about the patches for all debugging rounds. From left to right: number of patches overlaying the background with the inter-annotator agreement of at least 80%, number of patches on which the agreement is under 80%, number of patches activating on the bird with an agreement of at least 80% and the total number of patches shown to the participants.

<table>
<thead>
<tr>
<th>ROUND</th>
<th># BACKGROUND</th>
<th># BIRD</th>
<th># NO AGREEMENT</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>19</td>
<td>73</td>
<td>5</td>
<td>97</td>
</tr>
<tr>
<td>SECOND</td>
<td>6</td>
<td>63</td>
<td>5</td>
<td>74</td>
</tr>
<tr>
<td>THIRD</td>
<td>3</td>
<td>73</td>
<td>2</td>
<td>78</td>
</tr>
</tbody>
</table>

Your task will be to carefully inspect each image and then to determine whether the highlighted overlay covers some part of the bird or whether, instead, it covers exclusively (or very nearly so) the background.

Table 3 exposes statistics about user feedback over the debugging rounds. The number of patches on which the prototypes activate almost exclusively on the backgrounds decreases by giving more supervision to the model. The inter-annotation agreement is 96% and shows that the ProtoPDebug feedback required from the supervisor is simple and effective.