Project Report
LSP-309

Interpretable Convolutional Neural Networks and Adversarial Examples: FY19 Line-Supported Program

N. Kaushik
J.K. Su

18 December 2020

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

Interpretable Convolutional Neural Networks and Adversarial Examples: FY19 Line-Supported Program

N. Kaushik
J.K. Su
Group 45

Project Report LSP-309
18 December 2020

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

Convolutional neural networks (CNNs) can achieve remarkable accuracy on many computer-vision and image-recognition problems, but their prediction mechanism is difficult or impossible to understand and normal training produces models that are highly susceptible to adversarial examples —inputs designed by an attacker to be misclassified despite being visually indistinguishable from ordinary images of the correct class. These shortcomings have sparked great interest in explaining CNNs and in training CNNs that are resistant to adversarial inputs. Most research has investigated either explainability or robustness, but in this report, we show that one can apply robust optimization techniques to an interpretable CNN and train models that are both interpretable and robust.

We take a "prototype" CNN developed by Li *et al.* [1] that replaces the conventional classification layer with an interpretable classification layer derived from case-based reasoning principles. We compare and contrast the prototype CNN with a conventional, non-intepretable CNN and show some visualizations of its prediction process.

Then we review some common attacks, such as the fast gradient sign method (FGSM) and projected gradient descent (PGD), which can be used to generate adversarial examples. We show that, like a conventional CNN, a prototype CNN trained in the normal way is susceptible to adversarial inputs. Hence, we confirm that intepretability alone does not provide robustness. Nevertheless, some example visualizations help illustrate understand why a misclassification occurred. They also suggest that the interpretable classification layer might allow one to detect an adversarial input.

Next, we review robust optimization methods that a defender could employ to train neural networks to resist attacks. We also explain that the structure of the interpretable CNN leads to a variety of options for both the attacker and defender. These options make robust optimization training of the prototype CNN more complicated than for a conventional CNN.

Finally, we train prototype CNNs for each of the options, and we obtain trained models that are simultaneously interpretable and robust. Example visualizations indicate that the interpretable classification layer is unlikely to facilitate detection of adversarial inputs, but the robustness of these models makes this property less important. Overall, we show that robust optimization can produce trained prototype CNNs that are interpretable, unlike a conventional CNN, and robust, just like a similarly trained conventional CNN.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Page

1. INTRODUCTION AND MOTIVATION

Deep neural networks offer state-of-the-art performance on many machine-learning problems. For computer-vision tasks like image recognition and object detection, deep *convolutional neural networks* (CNNs) [2] have demonstrated predictive accuracy comparable to that of human experts [3] [4], and in some cases, they have achieved superhuman accuracy [5]. It has also been reported that early CNN layers learn low-level features like edges and patches of color, middle layers learn mid-level concepts like eyes or ears, and later layers learn high-level concepts like faces [6] [4].

Although neural networks can provide excellent predictive accuracy, their prediction mechanism is so complicated that humans, including machine-learning experts, cannot understand it. A neural network is often described as an opaque, impenetrable "black box" that outputs its prediction without any explanation or justification.

In addition, researchers have shown that a neural network can be fooled or tricked into making an incorrect prediction while still reporting high confidence in the prediction. In image recognition, a *fooling image* is a nonsense image (e.g., noise or meaningless patterns) that a neural network nonetheless confidently predicts to belong to a particular class [7]. More concerning, one can take an ordinary image that is properly classified by a neural network and make imperceptible modifications to it to create an *adversarial image* that is visually identical to the original yet confidently and incorrectly classified by the same neural network [8, 9]. Additional research has shown that these modifications can be made in the physical world rather than as perturbations of pixel values on a computer [10, 11].

These undesirable properties—an unexplainable prediction mechanism and susceptibility to fooling or adversarial inputs—have made many people distrustful of neural networks and unwilling to accept them for applications that can have serious consequences. However, the excellent predictive performance remains tantalizing, and unsurprisingly, research and interest have grown explosively in two different areas:

- **Explanation**: Understanding and explaining how a neural network reached its decision
- **Robustness**: Designing and training neural networks to be resistant to adversarial inputs

Although there is much work in each area treated separately, this report considers both areas simultaneously. We examine a CNN whose classification layer uses an *interpretable model*, meaning that it is deliberately designed to be explainable or understandable. It was developed by Li *et al.* [1], who were partially supported by this Line-supported program. We also apply techniques from robust optimization to train the CNN to be resistant to adversarial inputs [12]. In this way, we demonstrate that it is possible to develop neural networks that are both interpretable and resistant to adversarial inputs.

A large amount of work on explaining machine learning may be described as *post hoc* (Latin: "after the event"): one takes a trained, non-interpretable or blackbox model and tries to generate an approximate explanation for how it makes a prediction. One class of *post hoc* approaches fits a simpler model, which may be an interpretable model like a decision tree or linear classifier, to the

outputs of the blackbox model and uses the simpler model for the explanation [13]. For CNNs, another approach is deconvolution [6], which attempts to visualize the features that stimulate parts of the network. A wide variety of gradient-based techniques use a neural network's gradients and back propagation to generate "saliency maps" [14], which aim to identify input pixels that significantly affect or contribute to the network's prediction [15] [16] [17] [18] [19]. Another group of gradient and back propagation methods imposes a conservation principle on each neuron's contribution to the network's output [20] [21].

Although there are times when the *post hoc* approach to explanation can be useful, an inherent drawback of *post hoc* explanation is that it does not truly describe how the original model makes its predictions. As a result, a user may form an incorrect mental model for the prediction mechanism. For example, if one uses a simpler model to approximate the blackbox model, then discrepancies can arise when the original model and the simpler model make conflicting predictions. One might also argue that such discrepancies are inevitable; if they did not occur, then one could just use the simpler model instead of the blackbox model. Adebayo *et al.* [22] have shown that some saliency and gradient techniques can be misleading. In one experiment, they randomized the weights of an increasing number of layers in a trained network. One would expect saliency to decline as more layers' weights were randomized, but the saliency maps for some methods continued to highlight parts of the input image like edges and other features, even when the entire network was randomized. In another set of experiments, Adebayo *et al.* trained a network on randomized labels so that the trained model essentially made random guesses rather that feature-driven predictions. One would expect the corresponding saliency maps to be meaningless, but again the saliency maps for some techniques emphasized parts of the input images.

In contrast, if one adopts an interpretable model, then there is no need to come up with an approximate explanation for how predictions are made. We therefore advocate approaches to explanation that use interpretable models because they are faithful to the actual prediction process. Rudin presents additional arguments for using interpretable models [23]. The complexity of neural networks means that their feature-extraction layers might not be fully interpretable, but models like that of Li *et al.* [1] employ a classification layer that is interpretable. A related interpretable CNN is the "prototype-parts" CNN by Chen *et al.* [24]. It is a novel extension of the prototype CNN of Li *et al.* [1] that learns prototypical parts, rather than image-sized prototypes, and uses the detected parts to make its prediction; this network was also developed in part with support from this Line-supported program.

The rest of the report covers the following topics. Section 2 reviews a conventional CNN and the interpretable "prototype" CNN of Li *et al.* [1]. Section 3 presents some visualizations of the interpretable prediction process that can help one understand how the network makes its predictions. More important, in the case of a misclassification, they can help one understand why the mistake occurred.

Next, Section 4 introduces the concept of an attacker who wants to defeat a CNN by making adversarial examples that look like ordinary images but are misclassified. Section 5 examines how the prototype CNN performs when presented with adversarial examples. Although the network has not been trained to resist such inputs, we are curious whether its interpretable structure could

help one understand why an adversarial example was misclassified or even detect that an input is an adversarial example.

Section 6 presents the concept of a defender who wants to prevent misclassification of adversarial inputs. It discusses how robust optimization can be employed to train a CNN that is resistant to adversarial examples. The interpretable nature of the prototype network introduces some new attacker and defender considerations for robust optimization, so Section 7 describes a number of ways that robust optimization can be incorporated into training of the prototype network. Section 8 presents experimental results for prototype networks trained in these ways and subjected to adversarial examples. Finally, Section 9 offers a summary and conclusions.

This page intentionally left blank.

2. REVIEW OF CONVENTIONAL AND PROTOTYPE NETWORKS

This section reviews supervised classification, a conventional CNN, and the prototype neural network of Li *et al.* [1].

2.1 SUPERVISED CLASSIFICATION

Image recognition is a special case of *supervised classification*: given an image $\mathbf{x}$ that belongs to exactly one of K possible classes, the goal is to train and deploy a classifier that accurately predicts the correct class y to which $\mathbf{x}$ belongs. Each input image $\mathbf{x}$ has height H, width W, and D channels; e.g., $D = 1$ if the inputs are monochrome images, and $D = 3$ if they are RGB images. The number of channels is sometimes referred to as *depth*. An input image $\mathbf{x}$ can be considered either as a three-dimensional $H \times W \times D$ tensor or as a vector of length $P = H \times W \times D$; $\mathbf{x}$ is a member of the *input space* $\mathcal{X} \subseteq \mathbb{R}^P$. Without loss of generality, the set of classes can be represented as $\mathcal{Y} = \{1, 2, \ldots, K\}$. For example, recognition of animal images could use labels like "cat", "dog,", "horse," etc., and each label can be mapped to a corresponding integer in $\{1, 2, \ldots, K\}$. Let $y \in \mathcal{Y}$ be the correct class associated with $\mathbf{x}$.

In this preliminary discussion, we denote the classifier as a function $h_{\boldsymbol{\theta}} : \mathcal{X} \rightarrow \mathcal{Y}$ that is parameterized by $\boldsymbol{\theta}$. Given an input image $\mathbf{x}$ (but not the correct class y), the classifier returns a predicted class $\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x})$, with $\hat{y} \in \mathcal{Y}$. The prediction is correct if $\hat{y} = y$; otherwise, it is incorrect.

Training employs a training set $\mathcal{T}_{\text{train}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_{\text{train}}}$ that contains pairs $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is a training image, and y_i is the known, correct class associated with $\mathbf{x}_i$. Training learns the parameters $\boldsymbol{\theta}^*$ that optimize the accuracy of the classifier's predicted classes $\{\hat{y}_i = h_{\boldsymbol{\theta}^*}(\mathbf{x}_i) : \mathbf{x}_i \in \mathcal{T}_{\text{train}}\}$ by comparing the predictions against the corresponding correct classes from $\mathcal{T}_{\text{train}}$. The trained classifier is $h_{\boldsymbol{\theta}^*}$.

Because the correct classes are available to guide the training procedure, this approach to classification is said to be "supervised." A different form of classification, not covered in this report, is "unsupervised classification," in which information about the correct classes is not available during training or testing.

Following training, testing assesses the accuracy of the trained classifier on a separate testing set $\mathcal{T}_{\text{test}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_{\text{test}}}$. It is assumed that the pairs $(\mathbf{x}_i, y_i)$ in $\mathcal{T}_{\text{train}}$ and $\mathcal{T}_{\text{test}}$ are drawn from the same underlying distribution. The parameters $\boldsymbol{\theta}^*$ are now held fixed, and the predicted classes $\{\hat{y}_i = h_{\boldsymbol{\theta}^*}(\mathbf{x}_i) : \mathbf{x}_i \in \mathcal{T}_{\text{test}}\}$ are compared against the corresponding correct classes from $\mathcal{T}_{\text{test}}$ to measure the classifier's accuracy on the testing set. This accuracy is typically used as the final performance measure because it reflects the classifier's ability to generalize to data outside the training set.

In this work, we focus on a common benchmark image recognition dataset: the *Modified NIST* Set (MNIST) dataset, which consists of grayscale images of handwritten digits 0, 1, ..., 9 [25, Sec. III-A]. MNIST contains 55,000 training images and 10,000 test images, each of size 28×28 pixels. Given an image $\mathbf{x}$ of a handwritten digit, the goal is to predict the digit or class y of the image accurately.

2.2 CONVENTIONAL CNN

Here we review a conventional CNN, depicted in Figure 1. Tutorials on CNNs can be found in [26], [27], [28].

2.2.1 Architecture

The network consists of a number of convolutional layers followed by a fully connected layer. We use a tilde ($\tilde{}$) to indicate items associated with the conventional CNN. The input image $\mathbf{x}$ is passed into the convolutional layers, which are parameterized by weights $\tilde{\boldsymbol{\alpha}}$ and denoted by $\tilde{f}_{\tilde{\boldsymbol{\alpha}}}$. The convolutional layers apply a sequence of spatially localized nonlinear transformations that produce $\tilde{\mathbf{z}}$:

$$\tilde{\mathbf{z}} = \tilde{f}_{\tilde{\boldsymbol{\alpha}}}(\mathbf{x}). \tag{1}$$

$\tilde{\mathbf{z}}$ may be viewed as an $\tilde{H}_{\text{lat}} \times \tilde{W}_{\text{lat}} \times \tilde{D}_{\text{lat}}$ tensor or a vector of length $\tilde{Q} = \tilde{H}_{\text{lat}} \times \tilde{W}_{\text{lat}} \times \tilde{D}_{\text{lat}}$. It is a transformed version of $\mathbf{x}$ called a *latent vector* or feature vector. Mathematically, $\tilde{f}_{\tilde{\boldsymbol{\alpha}}} : \mathcal{X} \to \tilde{\mathcal{Z}}$, where $\tilde{\mathcal{Z}} \subseteq \mathbb{R}^{\tilde{Q}}$ is the network's *latent space*. Appendix 1 reviews the details of the processing in the convolutional layers; also see [28].

The latent representation $\tilde{\mathbf{z}}$ is then fed into the fully connected layer, whose parameters can be arranged as a $K \times \tilde{Q}$ weight matrix $\tilde{\mathbf{W}}$. This layer applies the following operations to obtain the predicted class $\hat{\tilde{y}}$:

$$\tilde{\mathbf{v}} = \tilde{\mathbf{W}}\tilde{\mathbf{z}}, \tag{2}$$

$$\tilde{\mathbf{q}} = \text{softmax}(\tilde{\mathbf{v}}), \tag{3}$$

$$\hat{\tilde{y}} = \arg\max_j \tilde{\mathbf{q}}[j]. \tag{4}$$

Equation (2) is just a linear transformation of $\tilde{\mathbf{z}}$. Each element $\tilde{\mathbf{v}}[j]$ is sometimes called a *logit*, so $\tilde{\mathbf{v}}$ is called the *logit vector* or *logits*, and it belongs to *logit space*, a subset of $\mathbb{R}^K$.

Next, for an input vector $\mathbf{w} \in \mathbb{R}^K$, the softmax operation in (3) is defined as

$$\mathbf{q} = \text{softmax}(\mathbf{w}), \quad \text{where} \quad \mathbf{q}[j] = \frac{\exp(\mathbf{w}[j])}{\sum_{k=1}^{K} \exp(\mathbf{w}[k])}, \quad j = 1, 2, \ldots, K. \tag{5}$$

This operation converts $\mathbf{w}$ into a length-K vector $\mathbf{q}$ that lies in the $(K-1)$ probability simplex $\mathbb{S}^{K-1} = \{\mathbf{q} \in \mathbb{R}^K : \mathbf{q}[j] \geq 0, \forall j; \sum_{j=1}^{K} \mathbf{q}[j] = 1\}$; that is, the elements of $\mathbf{q}$ are non-negative and sum to unity. Thus, each element $\tilde{\mathbf{q}}[j]$ of $\tilde{\mathbf{q}}$ represents the network's estimate of the posterior probability of class j given $\mathbf{x}$.

Finally, from (4), the predicted class $\hat{\tilde{y}}$ is the class that corresponds to the maximum of $\tilde{\mathbf{q}}$, and the estimated posterior probability is $\tilde{\mathbf{q}}[\hat{\tilde{y}}]$.

2.2.2 Training Objective for Conventional CNN

Because the softmax and arg-max operations have no parameters, the complete set of network parameters that must be learned during training is $\tilde{\boldsymbol{\theta}} = \{\tilde{\boldsymbol{\alpha}}, \tilde{\mathbf{W}}\}$. Let $\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\cdot)$ denote the processing

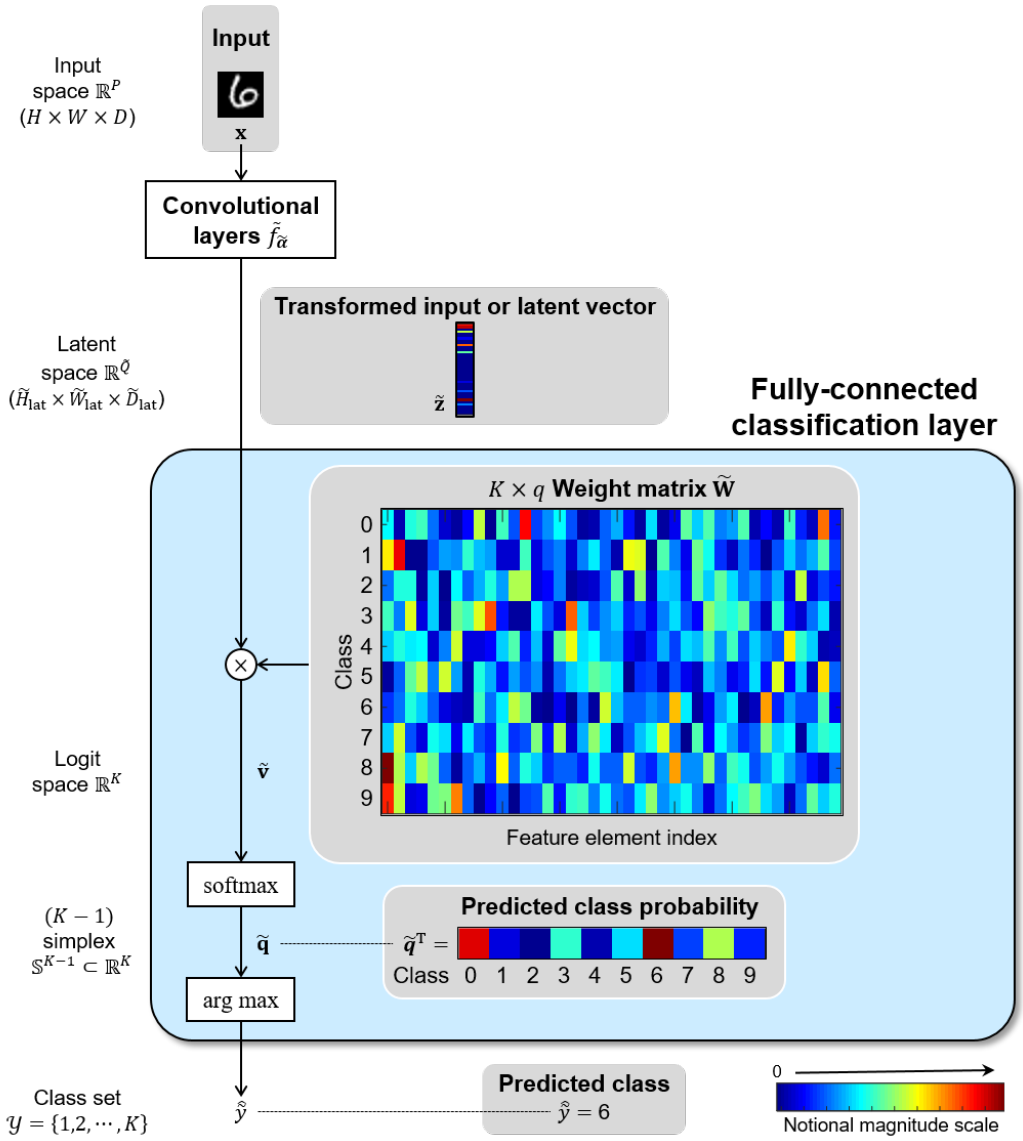


Figure 1. Architecture of a conventional CNN.

from the input through to the logits; that is,

$$\tilde{\mathbf{v}} = \tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}) \triangleq \tilde{\mathbf{W}} \times \tilde{f}_{\tilde{\boldsymbol{\alpha}}}(\mathbf{x}). \tag{6}$$

For multi-class classification, conventional training learns parameters that minimize the empirical loss on $\mathcal{T}_{\text{train}}$:

$$\min_{\tilde{\boldsymbol{\theta}}} \frac{1}{N} \sum_{i=1}^{N} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i), y_i\big), \tag{7}$$

where $L(\mathbf{v}, y) : \mathbb{R}^K \times \mathcal{Y} \to \mathbb{R}$ is the the *cross-entropy loss*:

$$L(\mathbf{v}, y) = -\log\left(\frac{\exp(\mathbf{v}[y])}{\sum_{k=1}^{K}\exp(\mathbf{v}[k])}\right) = -\mathbf{v}[y] + \log\left(\sum_{k=1}^{K}\exp(\mathbf{v}[k])\right).$$

The cross-entropy loss equals the negative logarithm of the yth element of softmax($\mathbf{v}$) [see (5)]. By minimizing the cross-entropy loss, training attempts to find the parameters $\tilde{\boldsymbol{\theta}}$ that maximize the log-probability of the correct class over the training set. Training is accomplished using a version of gradient descent and the back propagation procedure of recursively computing derivatives with respect to the parameters in $\tilde{\boldsymbol{\theta}}$.

2.2.3 Difficulties Interpreting CNNs

Although CNNs can achieve state-of-the-art and even superhuman predictive accuracy in some applications, they are notoriously difficult to understand. The series of nonlinear transformations in the convolutional layers mean that the latent space $\tilde{\mathcal{Z}}$ is difficult to visualize or understand, and a latent vector $\tilde{\mathbf{z}} \in \tilde{\mathcal{Z}}$ does not have any meaning to humans. In addition, the lack of understanding of the latent space also makes it impossible to comprehend the elements of the weight matrix $\tilde{\mathbf{W}}$.

2.3 PROTOTYPE NETWORK

Here we review the interpretable prototype network of Li *et al.* [1]; its architecture is shown in Figure 2.

2.3.1 Architecture

The network uses a *convolutional autoencoder* (CAE) [29], followed by an interpretable classification layer that employs a novel *prototype layer* and a fully connected layer.

The CAE includes an auto-encoder f with weights $\boldsymbol{\alpha}$ and a decoder g with weights $\boldsymbol{\beta}$. The autoencoder uses convolutional layers to transform an image into a latent vector (or $H_{\text{lat}} \times W_{\text{lat}} \times D_{\text{lat}}$ tensor) $\mathbf{z}$ in latent space $\mathcal{Z}$. These layers behave essentially the same as the convolutional layers $\tilde{f}_{\tilde{\boldsymbol{\alpha}}}$ in the conventional CNN. However, the additional decoder contains deconvolutional layers that can transform a latent vector $\mathbf{z}' \in \mathcal{Z}$ into an image. That is, $f_{\boldsymbol{\alpha}} : \mathcal{X} \to \mathcal{Z} \subseteq \mathbb{R}^Q$, $Q = H_{\text{lat}} \times W_{\text{lat}} \times D_{\text{lat}}$, and $g_{\boldsymbol{\beta}} : \mathcal{Z} \to \mathcal{X}$, or:

$$\mathbf{z} = f_{\boldsymbol{\alpha}}(\mathbf{x}), \tag{8}$$

$$\hat{\mathbf{x}}' = g_{\boldsymbol{\beta}}(\mathbf{z}'). \tag{9}$$

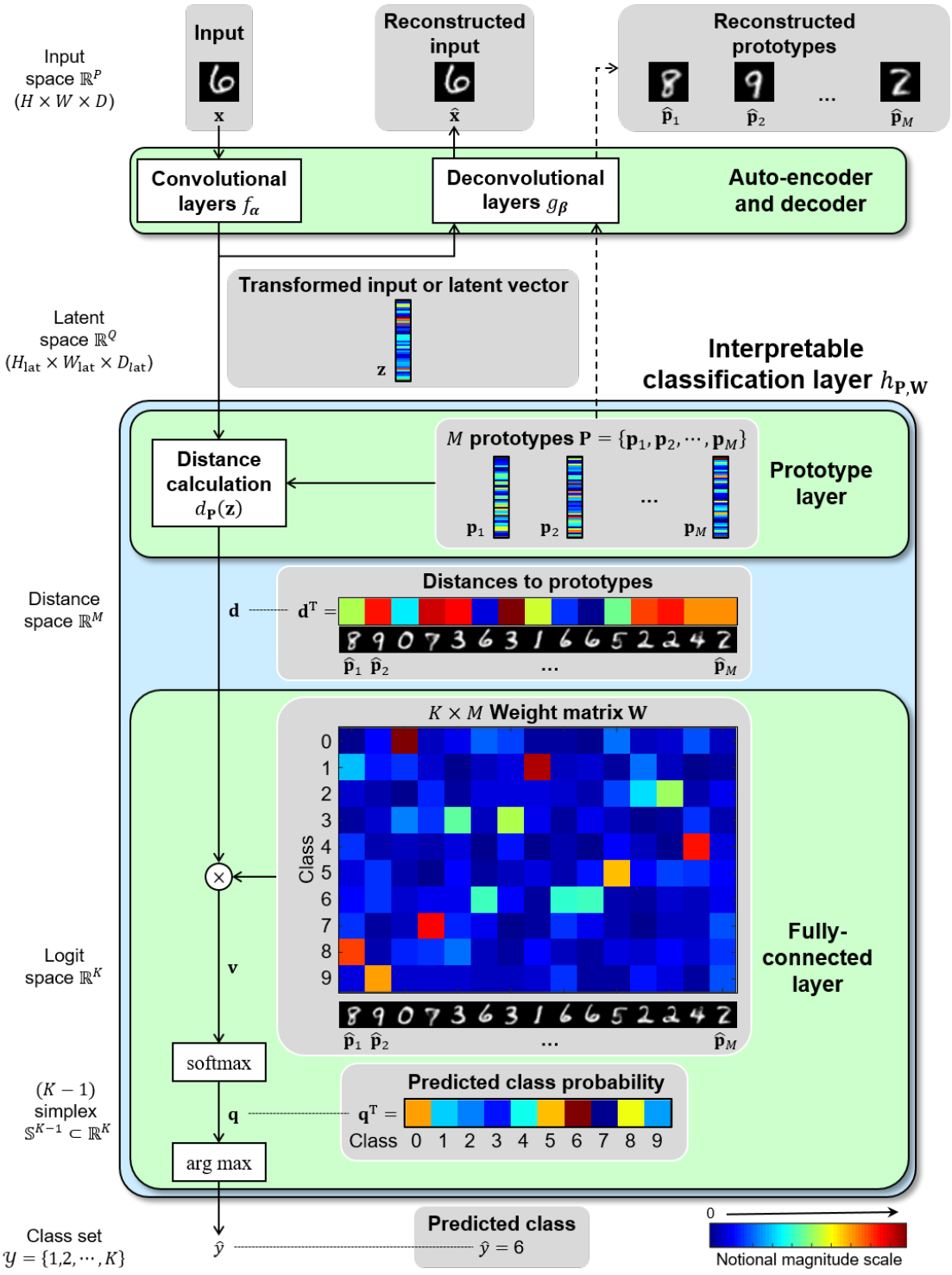


Figure 2. Architecture of interpretable prototype CNN.

During training, the weights $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are learned so that the reconstructed image $\hat{\mathbf{x}} = (g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x})$ is a good approximation to the original image $\mathbf{x}$. One benefit of the CAE is that a latent vector $\mathbf{z}'$ can be converted into a reconstructed image and visualized, which may be more meaningful than simply a point in an abstract, high-dimensional space.

The prototype layer includes a set $\mathbf{P} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_M\}$ of M latent vectors, which are referred to as *prototypes*. The prototypes can be thought of as synthetic exemplars that are learned during training (discussed below). Via the decoder, each *reconstructed prototype* $\hat{\mathbf{p}}_j = g_{\boldsymbol{\beta}}(\mathbf{p}_j)$, $j = 1, 2, \ldots, M$, can be visualized as an image. For each prototype $\mathbf{p}_j \in \mathbf{P}$, the prototype layer computes the distance between $\mathbf{z}$ and $\mathbf{p}_j$ to obtain the *prototype-distance vector* $\mathbf{d}$:

$$\mathbf{d} = \begin{bmatrix} \|\mathbf{z} - \mathbf{p}_1\|^2 \\ \|\mathbf{z} - \mathbf{p}_2\|^2 \\ \vdots \\ \|\mathbf{z} - \mathbf{p}_M\|^2 \end{bmatrix} = d_{\mathbf{P}}(\mathbf{z}). \tag{10}$$

(In the above equation, $d_{\mathbf{P}}(\mathbf{z})$ corresponds to $p(\mathbf{z})$ in Li *et al.* [1].) It can be convenient to convert $\mathbf{d}$ into an equivalent *similarity vector* $\mathbf{s}$ according to

$$\mathbf{s} = \begin{bmatrix} (\|\mathbf{z} - \mathbf{p}_1\|^2 + \varepsilon)^{-1} \\ (\|\mathbf{z} - \mathbf{p}_2\|^2 + \varepsilon)^{-1} \\ \vdots \\ (\|\mathbf{z} - \mathbf{p}_M\|^2 + \varepsilon)^{-1} \end{bmatrix} = \begin{bmatrix} 1/(\mathbf{d}[1] + \varepsilon) \\ 1/(\mathbf{d}[2] + \varepsilon) \\ \vdots \\ 1/(\mathbf{d}[M] + \varepsilon) \end{bmatrix}, \tag{11}$$

for some small $\varepsilon > 0$. Both $\mathbf{d}$ and $\mathbf{s}$ belong to $\mathbb{R}^M$; they quantify how close $\mathbf{z}$ is to each prototype.

The prototype-distance vector $\mathbf{d}$ is then passed to a fully connected layer with weight matrix $\mathbf{W}$ and processed much like in the conventional CNN:

$$\mathbf{v} = \mathbf{W}\mathbf{d}, \tag{12}$$

$$\mathbf{q} = \text{softmax}(\mathbf{v}), \tag{13}$$

$$\hat{y} = \arg\max_j \mathbf{q}[j]. \tag{14}$$

The crucial difference from the conventional CNN's fully connected layer is that the weight matrix is applied to the prototype-distance vector $\mathbf{d}$ rather than the latent vector $\mathbf{z}$. This modification means that the prototype CNN makes its prediction based upon how closely the input $\mathbf{x}$ resembles each of the M prototypes, in latent space.

The parameters of the interpretable classification layer are the set $\mathbf{P}$ of prototypes and the weight matrix $\mathbf{W}$. We denote the composition of (10) and (12) as $h_{\mathbf{P},\mathbf{W}}$, so the logits $\mathbf{v}$ are given by

$$\mathbf{v} = h_{\mathbf{P},\mathbf{W}}(\mathbf{z}) \stackrel{\triangle}{=} \mathbf{W} \times d_{\mathbf{P}}(\mathbf{z}). \tag{15}$$

2.3.2 Training Objective for Prototype CNN

For the prototype network, the number of prototypes is M, the parameters are $\boldsymbol{\theta} = \{\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\beta}\}$, and the training objective includes additional terms:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \left\{ \frac{1}{N}\sum_{i=1}^{N} L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(x_i), y_i\big) + \lambda_{\text{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(x_i), x_i\big) \right. \\ \left. + \lambda_1 \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(x_i)\big) + \lambda_2 \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(x_i), \mathbf{p}_j\big) \right\}, \quad (16)$$

Here, $d(x', x)$ denotes a distance measure in the *input space*, and $d_{\text{lat}}(z', z)$ denotes a distance measaure in the *latent space*. Typically, both use the L^2 norm, so $d(x', x) = \|x' - x\|$ and $d_{\text{lat}}(z', z) = \|z' - z\|_2^2$, but the distances are computed in different spaces and the distance measures could be different.

We can also express the objective as a weighted sum of individual terms that correspond to different constraints:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \big\{ L(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}) + \lambda_{\text{AE}} R_{\text{AE}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) + \lambda_1 R_1(\mathbf{P}, \boldsymbol{\alpha}) + \lambda_2 R_2(\mathbf{P}, \boldsymbol{\alpha}) \big\}, \quad (17)$$

where

$$L(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}) = \frac{1}{N}\sum_{i=1}^{N} L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(x_i), y_i\big) \qquad \text{cross-entropy loss} \quad (18)$$

$$R_{\text{AE}}(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(x_i), x_i\big) \qquad \text{autoencoder reconstruction error} \quad (19)$$

$$R_1(\mathbf{P}, \boldsymbol{\alpha}) = \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(x_i)\big) \qquad \text{prototype regularization term \#1} \quad (20)$$

$$R_2(\mathbf{P}, \boldsymbol{\alpha}) = \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(x_i), \mathbf{p}_j\big) \qquad \text{prototype regularization term \#2} \quad (21)$$

The cross-entropy loss is the usual loss function for optimizing predictive accuracy. The autoencoder reconstruction error is used to learn the decoder weights $\boldsymbol{\beta}$ so that a reconstructed (encoded and decoded) input resemble the original input. The first prototype regularization term encourages each prototype ($\mathbf{p}_j$) to be close to at least one training input (x_i), and the second prototype regularization term encourages each training input (x_i) to be close to at least one prototype ($\mathbf{p}_j$). The autoencoder reconstruction error is calculated in the ordinary input domain (e.g., pixel space for images), and the prototype regularization terms are calculated in the latent space.

2.3.3 Interpretability Advantages of the Prototype CNN

The prototype CNN was intentionally designed to include principles of *case-based reasoning* into the prediction process [30] [31]. The CAE allows the prototype CNN to learn useful features—

the prototypes—during training, which differentiates it from traditional case-based learning methods. Although the layers of the CAE are not completely interpretable, the classification layer has a number of appealing properties.

First, the decoder allows a vector $\mathbf{z}'$ in latent space to be reconstructed as an image that humans can better understand. In particular, the prototypes can be transformed from vectors in latent space into images that a human can readily view to understand what the prototypes look like. In this way, the reconstructed prototypes are like reference cases in case-based reasoning. Second, the prototype-distance vector $\mathbf{d}$ indicates the prototypes that the network considers similar to the latent-vector form of the input image. Third, the weights in the weight matrix $\mathbf{W}$ can be directly related to the input's similarity to a prototype and the classes that the prototype influences.

Overall, the prototype CNN builds interpretability into the network architecture and training objective, making *post hoc* explanation unnecessary. The prototype CNN truly makes its prediction based on the similarities between the transformed input image and the learned prototypes.

2.3.4 Dataset and Training Details

Like Li *et al.* [1], we train and test the prototype CNN on the MNIST dataset. The prototype network is trained with a learning rate of 0.0001 and with hyperparameters λ_{AE}, λ_1, and λ_2 set to 0.05. The autoencoder and prototype network are jointly optimized. A set of elastic deformations are used to augment the training data and reduce overfitting. Li *et al.* train the model for 1500 epochs and achieve a training accuracy of 99.53% and a test accuracy of 99.22%. We achieve consistent results in our replication study. Further details regarding data augmentation and training procedures can be found in [1].

In Section 3, we present visualizations of the trained prototype network. Section 4 discusses attacks and adversarial examples, and Section 5 examines the behavior of the prototype network on adversarial examples. Section 6 considers robust optimization methods for training classifiers to be resistant to adversarial examples. In Section 7, we explain ways to train and test the prototype network using robust optimization techniques; and in Section 8, we present results from such training.

3. VISUALIZATIONS FOR THE PROTOTYPE NETWORK

This section presents some simple visualizations for the basic prototype network to demonstrate its interpretability. We know from Li *et al.* [1] that the prototype network applied to the MNIST dataset achieves training and testing classification accuracies that are comparable with state-of-the-art CNNs. MNIST is indeed considered to be a "solved" dataset for which classifiers can often achieve 98+% accuracy, but its simplicity helps illustrate the advantages of using an interpretable CNN; namely, that such a network can provide a user with insight into how and why it made a particular prediction. This capability is useful both when the model makes a correct prediction and, perhaps more important, when it makes an incorrect prediction.

Because MNIST consists of a small number of very visually distinct classes, the network does not need to learn many prototypes to achieve high accuracy. This property allows us to visualize the network at each step during the inference process, from ingestion of an input image to generation of the final predicted probabilities for each class.

3.1 WEIGHT MATRIX

Table 1 shows the weight matrix in the classification layer of the trained prototype network. The 10 classes appear above the top row. The network decoder can reconstruct what each learned prototype looks like as an image, and the decoded, learned prototype images appear beside the leftmost column. The prototypes closely resemble handwritten digits, but an input's similarity to a particular prototype contributes to *all* classes. A more negative weight indicates a stronger contribution toward the corresponding class.

Typically, each prototype contributes most strongly to a single class, namely the one to which it bears the greatest visual similarity. For example, the second and fifth prototypes both resemble handwritten sevens, and both make their greatest contribution to class 7.

The third prototype offers an exception, as it contributes strongly to both class 4 and class 9. There is no other prototype that resembles a handwritten four or seven, which is consistent with the fact that training specifies the number of prototypes, but does not require that there be at least one prototype for each class.

3.2 CORRECT PREDICTION EXAMPLE

Figure 3 illustrates the prediction process for an example input that the model classifies correctly. The input image belongs to class 6. First, the network computes the similarities in latent space between the input image and each of the 15 learned prototypes. The leftmost bar chart in the figure shows the input's similarities to each of the decoded prototypes. Green highlighting shows that the input is most similar to three prototypes that visually resemble handwritten sixes.

Second, the network applies the learned weight matrix from Table 1 to calculate the logits for each class. Consulting the table shows that these three prototypes' weights (-5.15, -5.86, and

TABLE 1

Weight Matrix for the Trained Prototype Network

	0	1	2	3	4	5	6	7	8	9
	-0.70	0.27	1.55	0.69	0.20	0.59	-5.15	2.81	-1.66	1.58
	-0.58	2.57	2.44	2.11	-0.87	-0.80	1.94	-8.79	0.98	1.39
	3.59	3.45	0.87	2.10	-10.77	3.62	1.58	1.09	0.89	-10.00
	-0.02	-1.32	-7.47	-0.09	1.86	1.82	1.58	2.00	0.62	1.32
	0.68	-0.07	1.73	-1.88	2.57	1.48	2.57	-7.88	0.02	-0.68
	1.60	2.24	-8.19	-1.41	0.64	3.22	0.64	0.02	0.43	1.52
	1.65	-2.79	0.74	2.90	-2.40	-0.10	0.69	-1.28	-0.65	5.30
	1.11	0.22	1.06	-8.64	4.01	-0.63	1.23	0.46	1.57	-1.22
	-1.30	2.08	1.37	2.69	0.07	-0.97	-5.86	1.51	1.31	0.27
	0.79	1.54	2.43	0.50	2.14	1.21	2.58	1.14	-10.87	-0.77
	2.78	1.67	2.40	-2.07	1.98	-9.27	0.64	2.52	0.91	-2.16
	-14.11	0.95	0.36	2.47	1.54	2.06	1.54	0.74	2.11	-0.90
	1.59	-6.82	-0.19	1.02	-2.59	2.10	0.80	0.24	3.50	0.22
	0.31	1.18	0.48	0.08	-0.82	-1.22	-5.57	1.35	1.01	3.52
	2.00	-8.62	1.62	0.34	0.94	-0.11	0.40	1.96	1.38	1.37

Classes appear above the top row; decoded, learned prototype images appear next to the leftmost column. A more negative weight indicates a stronger contribution toward the corresponding class.

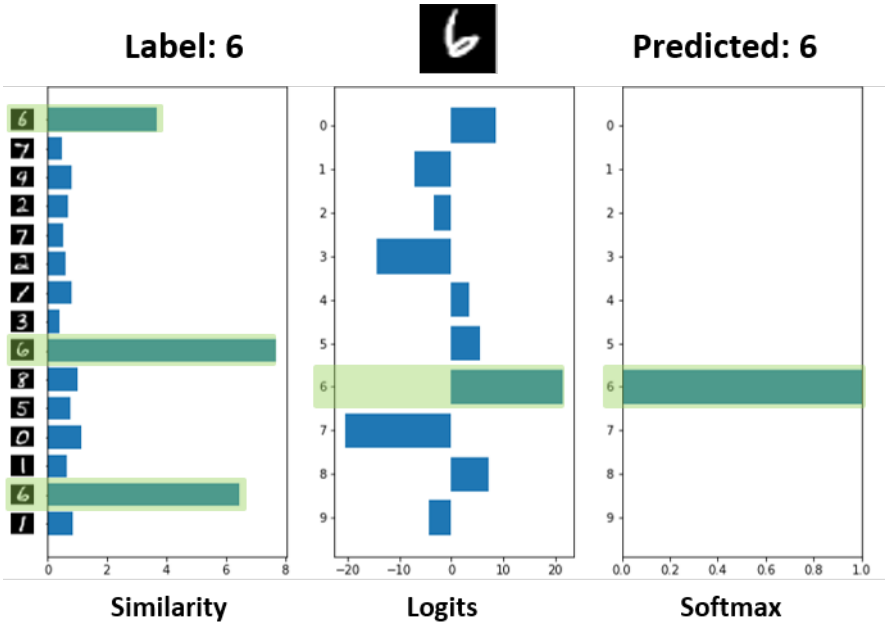


Figure 3. Example of the interpretable CNN's prediction for a correctly classified input.

-5.57), all contribute heavily in favor of class 6. The middle bar chart in the figure shows the resulting logits. Green highlighting shows that class 6 has the most positive logit value.

Finally, the softmax operation normalizes the logits to obtain predicted class probabilities, which appear in the rightmost bar chart in the figure. Green highlighting shows that the network obtains an overwhelming class probability in favor of class 6, so the network confidently (and correctly) predicts class 6.

This simple visualization is not possible with a conventional CNN. It effectively shows what the interpretable network is truly doing to make its prediction. The figure communicates to a user that:

1. The network considers the input image to be most similar to three prototypes whose decoded images resemble handwritten sixes.
2. After applying the weight matrix, the network finds that class 6 has the most-positive logit.
3. After softmax normalization, the network finds that class 6 is most probable, so it predicts class 6.

Stated more succinctly: "Because the input image is similar to the three prototypes that look like handwritten sixes, the network has decided to predict class 6."

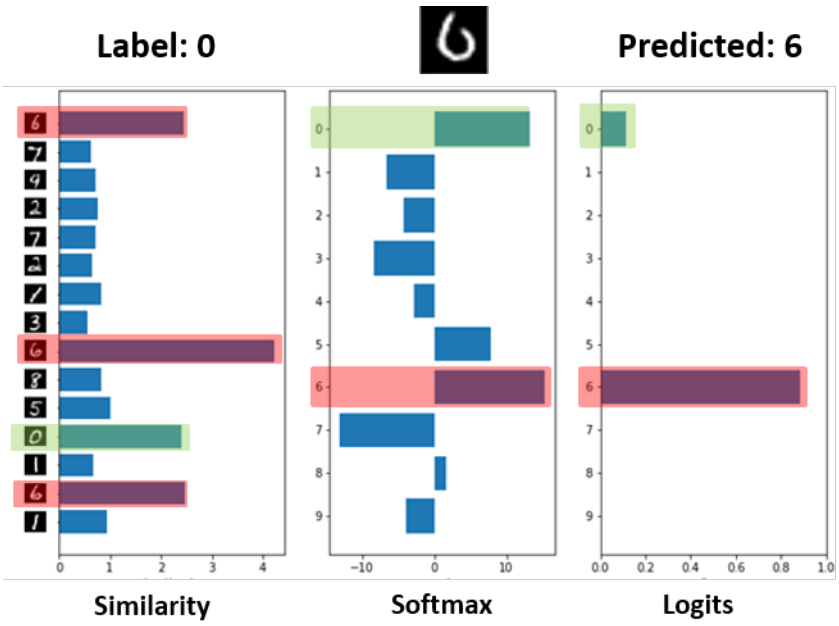


Figure 4. Example of the interpretable CNN's prediction for an incorrectly classified input.

3.3 INCORRECT PREDICTION EXAMPLE

Figure 4 shows an instance where the model makes an incorrect prediction. The example input belongs to class 0, but the handwritten image is an open loop, and the right part of the loop is much lower than the left part. Visually, the image looks like it could belong to either class 0 or class 6. The interpretable structure of the prototype network allows one to understand where and what went awry in the model's prediction process to cause this misclassification.

The leftmost bar graph displays the input's similarities to the prototypes. Green highlighting shows that the model considers the input to be quite similar to prototype #12, which resembles a handwritten zero. However, red highlighting shows that the model also considers the input similar to the three prototypes (#1, #9, #14) that resemble handwritten sixes. The similarities to prototypes #1 and #14 are comparable to the similarity to prototype #12, and the similarity to prototype #9 is much greater than the similarity to prototype #12.

From Table 1, the weight for prototype #12 and class 0 is -14.11, so the similarity to prototype #12 should produce a sizeable logit.[1] The logits appear in the middle bar chart, where green highlighting indeed shows a substantial logit value for class 0. However, red highlighting indicates a slightly larger logit value in favor of class 6.

Finally, the right bar chart displays the result of the softmax step. Although the logits for class 0 and class 6 are on par with one another, the softmax calculation amplifies the class with the largest logit and attentuates the other classes. Green highlighting shows that class 0 has a

[1] The weights for prototypes #1, #9, and #14 are the same as before.

non-negligible predicted probability, and red highlighting shows that class 6 receives the lion's share of the available class probability. Ultimately, the model ascribes higher confidence to the input belonging to class 6 than to class 0, so it (incorrectly) predicts class 6.

In this example, the model makes an incorrect prediction, but the simple visualization can help a user understand how and why the network made this choice.

1. The network finds that the input image is similar to prototypes that resemble either a hand-written zero or six, so there is already some ambiguity.
2. After applying the weight matrix, the logits for classes 0 and 6 are comparable, so ambiguity between these classes remains.
3. The softmax normalization tamps down the probability of class 0 and amplifies the probability of class 6, leading to the prediction of class 6.

A user can see that the network did not fail egregiously. It calculated reasonable similarities for the prototypes that visually resemble the input image, and its calculated logits show ambiguity between predicting class 0 or class 6. Because the logit for class 6 is largest, the model will definitely predict class 6, but because of the "winner-take-all" nature of the softmax operation, the network (incorrectly) predicts class 6 with a large predicted class probability.

Unlike a conventional, blackbox CNN, the interpretable prototype network offers the user a view into how it makes predictions, which can act like a sanity check and build user trust in the network's decision-making process.

This page intentionally left blank.

4. ATTACKS AND ADVERSARIAL EXAMPLES

This section reviews attacks that generate adversarial examples. Conceptually, one can imagine that there exists an *attacker*, who intentionally designs inputs to a machine learning model that fool the network or otherwise cause it to behave aberrantly [7]. Such inputs are called *adversarial examples* or *adversarial inputs*. The usual inputs, which have not been modified or designed in this way, are called *ordinary inputs*.

For an ordinary input $\mathbf{x}$ with correct label y, the attacker wishes to create an adversarial input $\mathbf{x}'$ that resembles $\mathbf{x}$ but is misclassified by the neural network. Let $\Delta(\mathbf{x})$ denote a neighborhood around $\mathbf{x}$ such that, if $\boldsymbol{\delta} \in \Delta(\mathbf{x})$, then $\mathbf{x}+\boldsymbol{\delta}$ still resembles $\mathbf{x}$ and one would expect $\mathbf{x}+\boldsymbol{\delta}$ to belong to class y. Typically, $\Delta(\mathbf{x})$ is an ε-ball under the ℓ_∞ norm in input space, which we denote by $\Delta_\varepsilon(\mathbf{x})$. Mathematically, for an ordinary input $\mathbf{x}$, the attacker wants to choose a perturbation $\boldsymbol{\delta}$ to produce an adversarial example $\mathbf{x}+\boldsymbol{\delta} \in \mathcal{X}$ that maximizes the cross-entropy loss $L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}+\boldsymbol{\delta}), y)$.

For an individual ordinary input $\mathbf{x}$, the adversarial input $\mathbf{x}'$ produced by an attack might not be misclassified because the attack parameters might not allow sufficient perturbation of $\mathbf{x}$. The effectiveness of an attack is assessed by applying it to a set of ordinary inputs to produce a corresponding set of adversarial inputs and examining the classifier performance on the latter set.

4.1 FAST GRADIENT SIGN METHOD (FGSM)

The *fast gradient sign method* (FGSM) is a simple, popular attack [9]. The attacker seeks a perturbation $\boldsymbol{\delta} \in \Delta_\varepsilon(\mathbf{x})$ such that $\mathbf{x}' = \mathbf{x}+\boldsymbol{\delta}$ maximizes the loss $L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}), y\big)$. FGSM exploits the fact that the gradient of the loss, calculated at $(\mathbf{x}, y)$ and with respect to the components of the input space $\mathcal{X}$, corresponds to the direction of the greatest increase in the loss. Consequently, the optimal perturbation is $\boldsymbol{\delta} = \varepsilon \operatorname{sign}\big[\nabla_{\mathcal{X}} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}), y\big)\big]$, and FGSM computes the adversarial input according to

$$\mathbf{x}'_{\mathrm{FGSM}} = \mathbf{x} + \varepsilon \operatorname{sign}\big[\nabla_{\mathcal{X}} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}), y\big)\big]. \tag{22}$$

In implementation, the gradient is computed via back propagation.

4.2 PROJECTED GRADIENT DESCENT (PGD)

FGSM takes a single step of size ε from $\mathbf{x}$ in the direction of greatest increase in the loss function. However, the shape of the loss surface can change rapidly, even in a small neighborhood. A more effective attack can be realized if one uses $\alpha \ll \varepsilon$ and repeats FGSM several times, each time recalculating the gradient and projecting the result to $\Delta_\varepsilon(\mathbf{x})$. This attack is known as *projected gradient descent* (PGD) or *iterated FGSM* (iFGSM) [32].

Given $\mathbf{x}$ and y, the attack initializes $\mathbf{x}'_0 = \mathbf{x}$, and then it calculates

$$\mathbf{x}'_n = \Pi_{\Delta_\varepsilon(\mathbf{x})}\Big(\mathbf{x}'_{n-1} + \alpha \operatorname{sign}\big[\nabla_{\mathcal{X}} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}'_{n-1}), y\big)\big]\Big), \quad n = 1, 2, \ldots, N_{\mathrm{PGD}}, \tag{23}$$

where $\Pi_{\Delta_\varepsilon(\mathbf{x})}(\mathbf{x}'')$ is the projection operator that clips $\mathbf{x}''$ to $\Delta_\varepsilon(\mathbf{x})$, and N_{PGD} is the number of steps. The resulting adversarial input is $\mathbf{x}'_{\mathrm{PGD}} = \mathbf{x}'_{N_{\mathrm{PGD}}}$.

4.3 TARGETED ATTACKS

This work did not use targeted attacks, but they are worthy of mention and have been included for completeness. The preceding attacks try to cause $\mathbf{x}$ to be misclassified by perturbing $\mathbf{x}$ away from the correct class y. There is no control over the misclassified predicted class for $\mathbf{x}'$. The attacks can be modified to push $\mathbf{x}$ toward a desired *target class* $y_{\text{tgt}} \neq y$ and cause $\mathbf{x}'$ to be misclassified as y_{tgt} [32].

In such targeted attacks, the loss is calculated for y_{tgt} rather than y, and each perturbation uses the negative of the gradient, which is the direction of greatest decrease in the loss for the target class. Hence, targeted FGSM uses

$$\mathbf{x}'_{\text{FGSM}} = \mathbf{x} - \varepsilon \operatorname{sign}\left[\nabla_{\mathcal{X}} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}), y_{\text{tgt}}\big)\right],$$

and targeted PGD uses

$$\mathbf{x}'_n = \Pi_{\Delta_\varepsilon(\mathbf{x})}\Big(\mathbf{x}'_{n-1} - \alpha \operatorname{sign}\left[\nabla_{\mathcal{X}} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}'_{n-1}), y_{\text{tgt}}\big)\right]\Big), \quad n = 1, 2, \ldots, N_{\text{PGD}}.$$

TABLE 2

MNIST Test Set Accuracy of Conventional CNN and Interpretable Prototype CNN, Trained in the Ordinary Manner, and Tested either on Ordinary Input Images or on Adversarial Input Images Generated by FGSM with $\varepsilon = 0.3$

Input Images	Type of CNN	Test Set Accuracy
Ordinary	Conventional	0.994
Ordinary	Prototype	0.991
Adversarial	Conventional	0.093
Adversarial	Prototype	0.112

5. ADVERSARIAL EXAMPLES AND THE PROTOTYPE NETWORK

We do not have any expectation that the prototype network should offer any inherent robustness against adversarial inputs, but we wanted to confirm this expectation. This section presents results when the FGSM attack of Section 4.1 is applied to a conventional CNN and the prototype CNN.

5.1 ACCURACY

We evaluate the performance of a standard CNN and the prototype CNN against adversarial images generated by the FGSM attack. We first train a standard CNN and the prototype CNN on ordinary (i.e., non-adversarial) images in the MNIST training set. We then use the FGSM attack with $\varepsilon = 0.3$ to perturb these same images, and evaluate both models' performance against this new adversarial test set. Table 2 displays the accuracy of the standard and prototype networks when evaluated on a test set of ordinary and adversarial images. When classifier performance is evaluated on a test set of ordinary images, both models perform equally well, and achieve up to 99.6% accuracy. Although the adversarial images closely resemble ordinary MNIST digits, neither model achieves better than random accuracy (i.e., 10% given that there are 10 classes).

These results confirm that the prototype network is not *automatically* more robust against adversarial inputs than a conventional CNN. It would have been an impressive additional benefit if the interpretable CNN's reliance on prototype similarities had also constrained the errors caused by perturbations in the decision space, but this is not so. Ultimately, this finding is not surprising given that the prototype network was not specifically designed or trained to be resistant to adversarial inputs.

5.2 VISUALIZATION

Although the prototype network is not inherently resistant to adversarial inputs, the interpretable nature of this model nonetheless allows a user to see *why* it misclassifies an adversarial example, which can provides some useful insights about the nature of FGSM attacks.

Figures 5 and 6 visualize the model's internal inference process when classifying an ordinary input image ("Real") and adversarial versions of the same image ("FGSM") generated with FGSM at $\varepsilon = 0.1$ and $\varepsilon = 0.3$, respectively. The three input images appear very visually similar, and even though the $\varepsilon = 0.3$ attack introduces some perceptible noise, a human user would still unmistakably understand them to all depict class 1.

The blue bars illustrate inference on the ordinary image, whereas the orange bars illustrate inference on the corresponding adversarial images. In both figures, the prototype network correctly classifies the ordinary image: the prototype similarity calculation indicates that the image is most similar to the two prototypes that resemble a handwritten one. These similarities are reflected downstream, as illustrated by the logits and softmax scores, which reveal the correct prediction is made with high confidence.

However, the figures show that the model misclassifies both adversarial images. It mistakes it as class 4 when $\varepsilon = 0.1$ and as class 8 when $\varepsilon = 0.3$. In latent space, where prototype distances are computed, the differences between ordinary and perturbed images are exploited by the attack. The visualizations show that the network considers the adversarial image more similar to the prototype that resembles a handwritten four than to those that resemble a handwritten one. The FGSM attack also causes the similarities to become "flattened"—smaller and spread almost uniformly among the prototypes. When the weight matrix is applied to the prototype similarities, the logit for class 4 becomes larger than the logits for the other classes, and the logit for correct class 1 becomes negative. Finally, the network incorrectly predicts class 4.

The FGSM attack with $\varepsilon = 0.3$ causes similar behavior. The adversarial image is noisier than at $\varepsilon = 0.1$, but still recognizable as class 1. However, the prototype similarities are now almost uniformly distributed over the prototypes. The logit for correct class 1 becomes most negative, and the logits for classes 8 and 9 are the largest ones and comparable in magnitude. The softmax results show the network considers the input as belonging to class 8, with class 9 as the next possible candidate. The network incorrectly predicts class 8.

5.3 POTENTIAL DETECTION OF ADVERSARIAL INPUTS

While conducting this preliminary inquiry, we observed another interesting side benefit of the prototype network. Although the model cannot correctly *classify* adversarial examples, its autoencoder and decoder might enable their *detection*.

As illustrated in Figure 7, FGSM-generated adversarial inputs that are encoded and decoded by the network have reconstruction errors that tend to be larger than those for ordinary inputs. The left side of the figure shows three corresponding input images (ordinary and adversarial) in the "Input" row, and their reconstructed versions from the prototype network's autoencoder and

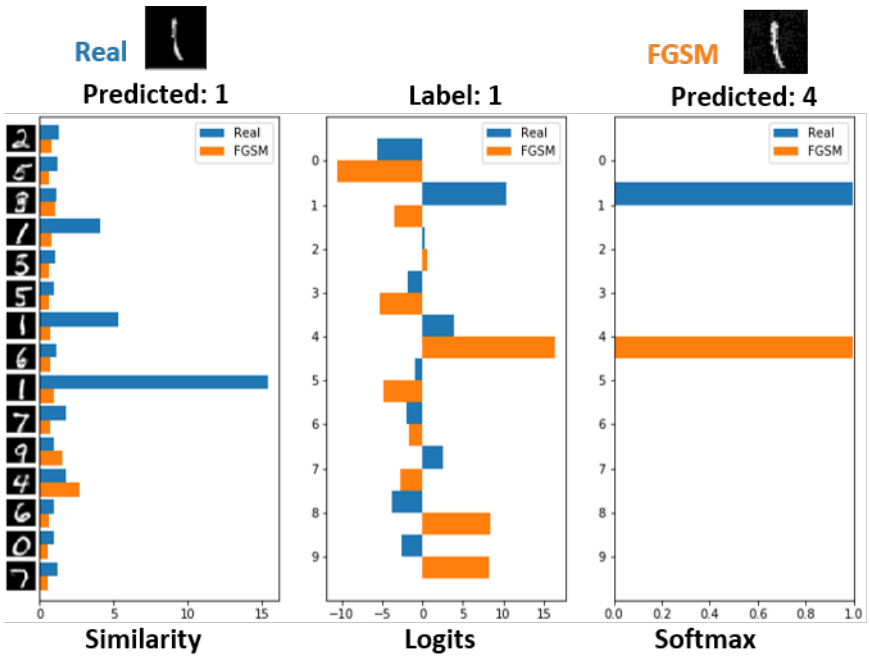


Figure 5. Example of the prototype network's prediction process for an ordinary image ("Real" image and blue bars) and an adversarial version of the same image ("FGSM" image and orange bars) produced by the FGSM attack with $\varepsilon = 0.1$. The predicted class for the ordinary image appears below the "Real" image, the correct class label is given as "Label," and the predicted class for the adversarial image appears below the "FGSM" image.

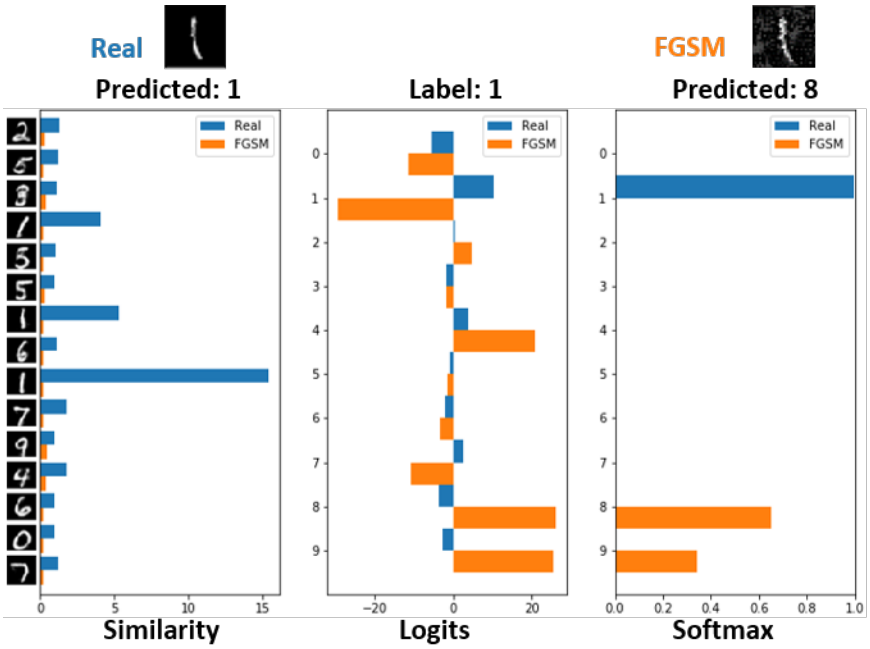


Figure 6. Example of the prototype network's prediction process for an ordinary image ("Real" image and blue bars) and an adversarial version of the same image ("FGSM" image and orange bars) produced by the FGSM attack with $\varepsilon = 0.3$. The predicted class for the ordinary image appears below the "Real" image, the correct class label is given as "Label," and the predicted class for the adversarial image appears below the "FGSM" image.

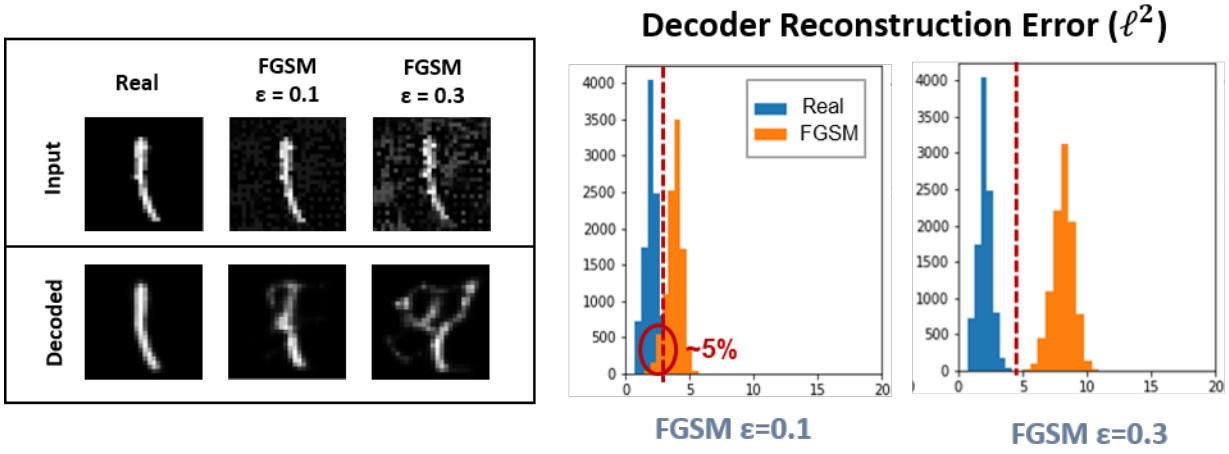


Figure 7. Decoder error for prototype network on ordinary an adversarial inputs. Left, top row: example input images (ordinary and for two FGSM attacks on the ordinary image). Left, bottom row: corresponding reconstructed images from the network's auto-encoder and decoder. Right: histograms of reconstruction error for ordinary (blue) and adversarial (orange) images for the two FGSM attacks.

decoder in the "Decoded" row. The decoded ordinary image closely resembles the initial ordinary image, but the decoded adversarial images are noticeably distorted compared to their initial images. The right side of the figure shows histograms of the reconstruction errors for ordinary images ("Real," blue) and adversarial images ("FGSM," orange).

The histograms suggest that it may be possible to distinguish between an ordinary image and an adversarial one 95% of the time for the FGSM attack with $\varepsilon = 0.1$. When ε is increased to 0.3, then it appears possible to detect an adversarial image every time.

Although the prototype network was not designed with the purpose of enabling detection of adversarial inputs, this observation is an interesting finding that indicates an alternative way it could be used to provide some ability to detect adversarial inputs, even though it is not resistant to them. Also, Figures 5 and 6 indicated that the distribution of the prototype similarities tends to favor a few prototypes when the input is an ordinary image and becomes flatter and more uniform when the input is an adversarial image. This property might also be exploited to detect adversarial inputs.

However, in these experiments, the FGSM attack was not trying to deceive the *decoder*, only the *classifier*. If the attacker modified the FGSM objective to account for the reconstruction error as well as the cross-entropy loss, then these potential detection capabilities of the prototype network might be diminished. In subsequent sections, we investigate the effects of adding this additional constraint to attacks while also employing robust optimization methods during network training.

This page intentionally left blank.

6. ROBUST OPTIMIZATION

This section reviews neural network training that uses robust optimization to try to make the trained network resistant to adversarial inputs. Tutorials on robust optimization of neural networks can be found in the article by Madry *et al.* [12] and on the web page for the *NeurIPS 2018* tutorial on adversarial robustness by Kolter and Madry [33].

6.1 ROLES OF ATTACKER AND DEFENDER

It is natural to think of an *attacker*, who generates adversarial examples to confuse the neural network, and a *defender*, who trains the neural network to resist the adversarial inputs.

Recall from Section 4 that the attacker wishes to produce adversarial inputs, slightly perturbed versions of ordinary inputs that the network misclassifies. For an ordinary input $\mathbf{x}$, the attacker constrains the adversarial input $\mathbf{x}' = \mathbf{x} + \boldsymbol{\delta}$ to lie in $\Delta_\varepsilon(\mathbf{x})$ so that it still resembles $\mathbf{x}$. Traditional, "undefended" training of a conventional CNN (see Section 2.2.2), attempts to minimize the empirical cross-entropy loss on $\mathcal{T}_{\text{train}}$ [see (7)]:

$$\min_{\tilde{\boldsymbol{\theta}}} \frac{1}{N} \sum_{i=1}^{N} L\big(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i), y_i\big).$$

Therefore, the attacker tries to *maximize* the loss function for $\mathbf{x}$; that is, the attacker seeks:

$$\max_{\boldsymbol{\delta} \in \Delta_\varepsilon(\mathbf{x})} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x} + \boldsymbol{\delta}), y). \tag{24}$$

FGSM and PGD in Section 4 are attacks that compute solutions—adversarial inputs $\mathbf{x}' = \mathbf{x} + \boldsymbol{\delta}$—to this equation.

Robust optimization introduces a defender, who wants the network to correctly classify not only ordinary inputs but also perturbed, adversarial versions of them. Doing so results in a different training objective. The defender should choose the network parameters $\tilde{\boldsymbol{\theta}}$ to minimize the result of (24) rather than the usual "undefended" loss $L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{w}), y)$. Mathematically, robust optimization aims to solve a saddle-point problem [12]:

$$\min_{\tilde{\boldsymbol{\theta}}} \frac{1}{N} \sum_{i=1}^{N} \max_{\boldsymbol{\delta} \in \Delta_\varepsilon(\mathbf{x}_i)} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i + \boldsymbol{\delta}), y_i). \tag{25}$$

If the defender succeeds, then the attacker must perturb an ordinary input $\mathbf{x}$ by such a large amount (i.e., $\boldsymbol{\delta} \notin \Delta_\varepsilon(\mathbf{x})$) that the adversarial input $\mathbf{x} + \boldsymbol{\delta}$ no longer resembles the class y associated with $\mathbf{x}$, and it will not be surprising or unsettling if the network predicts that $\mathbf{x} + \boldsymbol{\delta}$ belongs to some class other than y. In this way, the network might become resistant to adversarial perturbations inside $\Delta_\varepsilon(\mathbf{x})$.

6.2 GAME-THEORETIC PERSPECTIVE

The situation can be viewed from the perspective of game theory [34]. The attacker and defender are players competing in a game, where each player wishes to optimize its respective utility or penalty function, and each player does so by selecting the parameters under its control.

Let J_a denote the attacker's utility function. The attacker wants to maximize J_a by perturbing ordinary inputs subject to a constraint on the allowable perturbations. Let the set of perturbations over $\mathcal{T}_{\text{train}}$ be $\boldsymbol{\Delta} = \{\boldsymbol{\delta}_i\}_{i=1}^N$. In this context, $J_a(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta})$ is the cross-entropy loss:

$$J_a(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta}) = \frac{1}{N} \sum_{i=1}^{N} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i + \boldsymbol{\delta}_i), y_i), \tag{26}$$

and the attacker seeks

$$\begin{aligned} \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} J_a(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta}) &= \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} \frac{1}{N} \sum_{i=1}^{N} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i + \boldsymbol{\delta}_i), y_i) \\ &= \frac{1}{N} \sum_{i=1}^{N} \max_{\boldsymbol{\delta} \in \Delta_\varepsilon(\mathbf{x}_i)} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x}_i + \boldsymbol{\delta}), y_i), \end{aligned}$$

which corresponds to the averaged inner maximization in (25). The attacker cannot change the network parameters $\tilde{\boldsymbol{\theta}}$, but they appear in the utility function because the defender will optimize them.

Let J_d be the defender's penalty function. The defender wants to minimize J_d by choosing the network parameters $\tilde{\boldsymbol{\theta}}$. The penalty is equal to the maximum of the attacker's utility function, so

$$J_d(\tilde{\boldsymbol{\theta}}) = \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} J_a(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta}). \tag{27}$$

The penalty function does not depend on $\boldsymbol{\Delta}$ because the perturbations are marginalized out by the maximization operation. The defender seeks

$$\begin{aligned} \min_{\tilde{\boldsymbol{\theta}}} J_d(\tilde{\boldsymbol{\theta}}) &= \min_{\tilde{\boldsymbol{\theta}}} \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} J_a(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta}) \\ &= \min_{\tilde{\boldsymbol{\theta}}} \frac{1}{N} \sum_{i=1}^{N} \max_{\boldsymbol{\delta} \in \Delta_\varepsilon(\mathbf{x}_i)} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(x_i + \boldsymbol{\delta}), y_i), \end{aligned}$$

which corresponds to (25).

For this situation, we could have simplified the game-theoretic perspective and declared a zero-sum game with a single value function $J(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta})$ equal to the cross-entropy loss: $J(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta}) = \frac{1}{N} \sum_{i=1}^{N} L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(x_i + \boldsymbol{\delta}), y_i)$. The defender would minimize the value while the attacker would maximize it, which would give $\min_{\tilde{\boldsymbol{\theta}}} \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} J(\tilde{\boldsymbol{\theta}}, \boldsymbol{\Delta})$ and lead to (25). The utility and penalty functions become more useful when we robustly optimize the prototype network against adversarial inputs.

6.3 OPTIMIZATION APPROACHES

Kolter and Madry [33, Chapter 3 – Adversarial examples, solving the inner maximization; Strategies for the inner maximization] explain that there are three approaches for solving the inner maximization in (25); that is, solving (24).

The first approach is to find a lower bound for (24). If one empirically computes an adversarial example $\mathbf{x} + \boldsymbol{\delta}$ for $\mathbf{x}$, then calculating $L(\tilde{H}_{\tilde{\boldsymbol{\theta}}}(\mathbf{x} + \boldsymbol{\delta}), y)$ immediately provides a lower bound for the maximization. Of course, if the adversarial example is chosen poorly, then the bound might be loose and not very effective for robust optimization, so one should try to find a strong adversarial example. This rationale motivates the FGSM and PGD attacks, which each find the best lower bound within a class of adversarial examples. The empirical approach is the most common one in the literature and the one adopted in this work.

A second approach is to solve (24) exactly. For small networks with certain activation functions, like the rectified linear unit (ReLU), one can express the maximization as a mixed integer programming problem that can be solved exactly. This approach is important because it shows that, in some cases, an exact solution is possible, but these techniques do not scale for large networks.

The third approach is to find an upper bound for (24). Typically, one does not find an actual adversarial example, but applies techniques like convex relaxation to establish a guarantee that the solution to (24) does not exceed the calculated upper bound. The guarantee enables one to prove that a network is robust against attacks. This approach represents the state-of-the-art for robust optimization, but is much more complicated than the other approaches.

Finally, Madry *et al.* [12] and Schmidt *et al.* [35] recommend that a robust network be more expressive than an ordinary, non-robust one. For a conventional CNN, this requirement means that the network should have a larger number of parameters, such as more layers or a greater depth at each layer.

This page intentionally left blank.

7. ROBUST OPTIMIZATION OF THE PROTOTYPE NETWORK

The training objective for the prototype network includes additional terms besides the loss, so an immediate question is: How should one perform robust optimization of the prototype network? That is, how should one convert (16) or (17) into the form of (25)?

We consider both the attacker's and the defender's perspective to arrive at a number of possible forms.

7.1 ATTACKER'S PERSPECTIVE

The attacker wishes to cause misclassification, so the attacker should continue to increase the cross-entropy loss. However, if the attacker only maximizes the cross-entropy loss, then the reconstruction error might become large for adversarial examples while remaining small for non-adversarial ones. This property could enable the defender to detect whether an input is adversarial or not. Consequently, the attacker might also want to keep the reconstruction error small for adversarial inputs to reduce the effectiveness of such detection.

Regarding the prototype regularization terms, it is unclear whether an attacker would want to interfere with the prototypes because the attacker's main goal is to cause misclassification and perhaps evade detection rather than to make the prototypes similar to or dissimilar from adversarial inputs.

Consequently, there are two different forms for the inner maximization: [A1] one involving only the cross-entropy loss, and [A2] one involving a combination of the cross-entropy loss and the autoencoder reconstruction error. In the game-theoretic perspective of Section 6.2, the attacker has two possible utility functions. The first one is (26), the cross-entropy loss:

$$J_a(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\Delta}) = \frac{1}{N} \sum_{i=1}^{N} L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i + \boldsymbol{\delta}_i), y_i\big). \tag{28}$$

The second one reflects the attacker's desire to cause misclassification and keep the reconstruction error of adversarial inputs small; for some $\lambda'_{\mathrm{AE}} > 0$, it is given by

$$J_a(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\Delta}) = \frac{1}{N} \sum_{i=1}^{N} \Big[L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i + \boldsymbol{\delta}_i), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i + \boldsymbol{\delta}_i), \mathbf{x}_i + \boldsymbol{\delta}_i\big) \Big]. \tag{29}$$

7.2 DEFENDER'S PERSPECTIVE

For the defender, we present two different design approaches. We also point out that, for each ordinary training sample $\mathbf{x}_i$, the training procedure will generate a corresponding adversarial example, which we denote as $\mathbf{x}'_i$. Also, let $\mathbf{X}' = \{\mathbf{x}'_i\}_{i=1}^N$ be the set of adversarial training examples. Consequently, the training procedure knows whether an input is ordinary or adversarial, although the neural network does not.

In the first approach, the network employs *one set* of prototypes. The set might be larger than in the non-adversarial case, but one set of prototypes must handle both ordinary and adversarial inputs.

In the second approach, the defender introduces an additional set of *adversarial prototypes* specifically for addressing adversarial examples. They could help the network learn—and help a user understand—the decision boundaries between ordinary and adversarial inputs. Hence, the network has *two sets* of prototypes. The first set contains *ordinary prototypes* that are learned from ordinary inputs as usual, and the second set contains *adversarial prototypes* that are learned from the adversarial inputs in $\mathbf{X}'$.

For either approach, the defender continues to want to be robust to adversarial examples, so the defender continues to minimize the attacker's objective function. We discuss the other optimization terms for the two approaches next.

7.2.1 One Set of Prototypes

For a network with one set of prototypes, the defender has a few options regarding the autoencoder reconstruction error. First, the defender could minimize the autoencoder reconstruction error only for ordinary inputs, which might allow detection of adversarial inputs by examination of the reconstruction error. Second, the defender could extend the autoencoder reconstruction error term to include the adversarial inputs, which would put an additional burden on the autoencoder. However, this objective could make it more difficult to distinguish between ordinary and adversarial inputs, so we do not consider it. Third, the defender could attempt to make the autoencoder reconstruction error small for ordinary inputs, but large for adversarial inputs. Doing so would improve the ability to detect adversarial inputs, but it runs contrary to the purpose and intent of the autoencoder—particularly for adversarial inputs that closely resemble ordinary inputs—so we do not pursue it further.

Consequently, although we have described three possible options for the autoencoder reconstruction error, we only use the first option in the rest of this report.

For the two prototype regularization terms, there are two possibilities. One, indicated as [D1], is to apply the terms only to ordinary inputs so that the learned prototypes remain visually recognizable. In the game-theoretic perspective, this corresponds to the penalty function

$$J_d(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i \in \Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} \big[J_a(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\Delta})\big] + \lambda_{\text{AE}} \frac{1}{N} \sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big) + \lambda_1 \frac{1}{M} \sum_{j=1}^{M} \min_{i \in [1,N]} d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \lambda_2 \frac{1}{N} \sum_{i=1}^{N} \min_{j \in [1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big). \quad (30)$$

The maximization applies only to the attacker's utility function $J_a(\mathbf{P}, \mathbf{W}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\Delta})$.

The other possibility, indicated as [D2], is to apply the prototype regularization terms to both ordinary and adversarial inputs, which could diminish how recognizable the learned prototypes are.

This corresponds to the penalty function

$$J_d(\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}) = \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i\in\Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} \big[J_a(\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\Delta})\big] + \lambda_{\text{AE}}\frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}}\circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i),\mathbf{x}_i\big)$$
$$+ \lambda_1\frac{1}{M}\sum_{j=1}^{M}\min_{i\in[1,N]}\min_{\mathbf{x}_i,\mathbf{x}_i'}\Big[d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big), d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i')\big)\Big]$$
$$+ \frac{1}{2}\lambda_2\left[\frac{1}{N}\sum_{i=1}^{N}\min_{j\in[1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i),\mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N}\min_{j\in[1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i'),\mathbf{p}_j\big)\right]. \quad (31)$$

7.2.2 Two Sets of Prototypes: Ordinary and Adversarial

A network with M ordinary prototypes and M' adversarial prototypes is indicated as [D3]. For this option, let $\mathbf{P}$ be the set of ordinary prototypes as usual and $\mathbf{P'} = \{\mathbf{p}_1', \ldots, \mathbf{p}_{M'}'\}$ be the set of adversarial prototypes. The classification layer h includes $\mathbf{P'}$ as well as $\mathbf{P}$ and $\mathbf{W}$, where $\mathbf{W}$ is enlarged to include weights for the adversarial prototypes. The adversarial examples in $\mathbf{X'}$ can be used to learn the adversarial prototypes.

For the autoencoder reconstruction term, the options are identical to those for the network with one set of prototypes. Based on the same reasoning given above, we only consider the autoencoder reconstruction error for ordinary inputs.

With two sets of prototypes, it is natural to expand the prototype regularization terms so that each one contains separate terms for both ordinary and adversarial prototypes. As a result, the ordinary prototypes should remain visually recognizable, whereas the adversarial prototypes likely will not. This property could improve classification performance because the adversarial prototypes might help the network learn the decision boundaries between ordinary and adversarial inputs. It could also improve interpretability in two ways. First, because the adversarial prototypes are known to represent adversarial inputs instead of ordinary ones, a user could be comfortable with their lack of resemblance to recognizable inputs. Second, if the input is similar to adversarial prototypes and the network makes a mistake, a user might be more tolerant of such an error because it is evident that the input lies near the ordinary-versus-adversarial decision boundary.

This approach has the following penalty function:

$$J_d(\mathbf{P},\mathbf{P'},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}) = \max_{\boldsymbol{\Delta}:\{\boldsymbol{\delta}_i\in\Delta_\varepsilon(\mathbf{x}_i)\}_{i=1}^N} \big[J_a(\mathbf{P},\mathbf{P'},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta},\boldsymbol{\Delta})\big] + \lambda_{\text{AE}}\frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}}\circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i),\mathbf{x}_i\big)$$
$$+ \lambda_1\left[\frac{1}{M}\sum_{j=1}^{M}\min_{i\in[1,N]} d_{\text{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \frac{1}{M'}\sum_{j=1}^{M'}\min_{i\in[1,N]} d_{\text{lat}}\big(\mathbf{p}_j', f_{\boldsymbol{\alpha}}(\mathbf{x}_i')\big)\right]$$
$$+ \frac{1}{2}\lambda_2\left[\frac{1}{N}\sum_{i=1}^{N}\min_{j\in[1,M]} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i),\mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N}\min_{j\in[1,M']} d_{\text{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i'),\mathbf{p}_j'\big)\right]. \quad (32)$$

TABLE 3

Different Robust-Optimization Combinations for the Prototype Network

Attacker		Defender			Case & Equation
Opt.	Objective	Opt.	Prototype Set(s)	Prototype Regularization	
A1	Cross-entropy loss	D1	One	Ordinary	1 (B.33)
		D2		Ordinary and adversarial	2 (B.34)
		D3	Two: ordinary and adversarial	Separate ordinary and adversarial	3 (B.35)
A2	Cross-entropy loss and autoencoder reconstruction error	D1	One	Ordinary	4 (B.36)
		D2		Ordinary and adversarial	5 (B.37)
		D3	Two: ordinary and adversarial	Separate ordinary and adversarial	6 (B.38)

7.3 COMBINATIONS OF OPTIONS

The above discussion leads to two different attacker options: [A1] maximize the cross-entropy loss, or [A2] optimize a combination of the cross-entropy loss and autoencoder reconstruction error. It also produces three different defender options: [D1] use one set of prototypes and conduct prototype regularization for ordinary inputs; [D2] use one set of prototypes and conduct prototype regularization for ordinary and adversarial inputs; or [D3] use ordinary and adversarial prototypes and perform prototype regularization with separate terms for ordinary and adversarial inputs. In all cases, the defender calculates the autoencoder reconstruction error only over ordinary inputs.

The six different combinations of these options appear in Table 3, which includes references to the robust-optimization expressions that are given in Appendix 2.

8. RESULTS AND VISUALIZATIONS FOR THE ROBUSTLY OPTIMIZED PROTOTYPE NETWORK

We train and evaluate interpretable models using each of the six proposed combinations of attacker and defender strategies for robust optimization. We expect all of the models to provide some degree of robustness toward novel adversarial inputs given that they are exposed to these exemplars during the training process. We seek to understand if and why certain optimization strategies yield greater robustness than others, and to what extent model interpretability is preserved.

8.1 TRAINING DETAILS

We use the MNIST dataset again for model training and evaluation in the robust optimization setting. We train six separate prototype networks, one for each of the combined attacker and defender optimization strategies described in Table 3. During training, we employ the PGD attack with a design perturbation parameter $\varepsilon_d = 0.3$, step size $\alpha = 0.001$, and $N_{\text{PGD}} = 40$ iterations. Each batch of training images consists of an ordinary MNIST image $\mathbf{x}_i$ as well as its adversarial counterpart $\mathbf{x}'_i$. Both sets of images are used to learn the optimal prototypes and autoencoder, given the particular conditions of the objective function used.

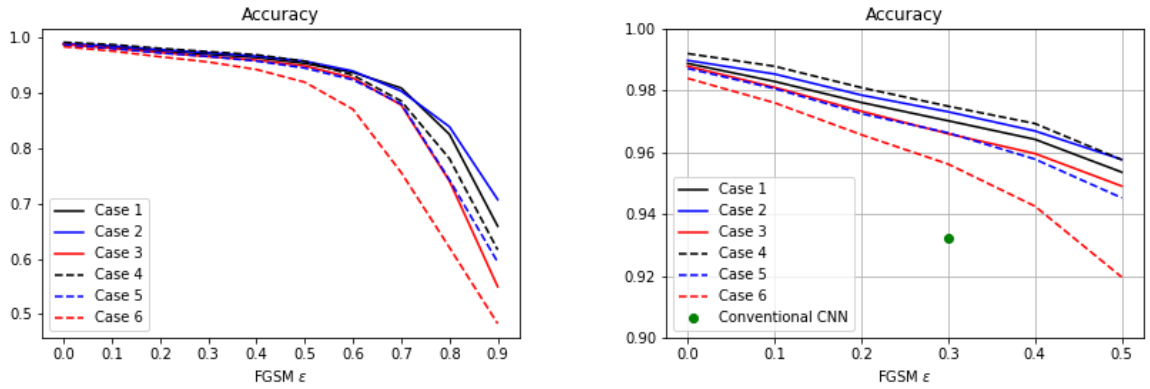
8.2 ACCURACY RESULTS

Accuracy results for the six robustly optimized prototype networks appear in Figure 8. The networks are evaluated on a test set of adversarial inputs generated via PGD with increasing levels of perturbation, from $\varepsilon = 0.1$ to 0.9 At $\varepsilon = 0$, the model operates only on unperturbed, ordinary images. All models demonstrate excellent accuracy on ordinary images. We expect this outcome given that all optimization strategies include the usual cross-entropy loss on ordinary images in their objective functions. More interestingly, all six strategies appear to produce models that are resilient against adversarial examples, up to the design $\varepsilon_d = 0.3$, and even beyond, in most cases.

To compare our results with those of a conventional CNN, we refer to results by Madry *et al.* [12, Table 1, Method: PGD, Source: A], who trained conventional CNNs on MNIST using robust optimization with PGD with $\varepsilon_d = 0.3$ and a variety of other settings. The robustly trained, conventional CNNs achieved accuracies between 89.3% and 93.2%. The value of 93.2% from [12, Table 1, Method: PGD, Steps: 40, Restarts: 1, Source: A] corresponds most closely with the attack we used for robust-optimization training of the prototype CNN. This operating point appears as a filled circle in Figure 8b. Our first conclusion is that it is indeed possible to train the interpretable prototype network to resist adversarial inputs.

8.3 PROSPECT OF DETECTING ADVERSARIAL INPUTS

In Section 5.3 and Figure 7, we discussed the possibility of using the decoder reconstruction error to detect an adversarial input. The initial results in that section were based on a model trained without robust optimization, with the FGSM attack used to form adversarial images.



(a) Broad view *(b) Detailed view*

Figure 8. Accuracy of robustly optimized networks when classifying adversarial images generated by FGSM attacks with increasing values of ε. The case numbers in the legend correspond to the cases in Table 3. Each network was robustly trained using adversarial examples generated by PGD with $\varepsilon_d = 0.3$.

In this section, training used robust optimization, and adversarial images were generated by the more powerful PGD attack. Figure 9 shows histograms of the reconstruction errors for ordinary and adversarial images for Case 1. The histograms are virtually identical, and we observed the same behavior for the other cases.

With the more powerful PGD attack, it is highly unlikely that a defender could use the reconstruction error to detect adversarial inputs, which obviates the initial findings from Section 5.3. An attacker using PGD does not need to worry about this avenue of detection, which reduces the motivation for Cases 4–6.

8.4 PROTOTYPE VISUALIZATIONS

Visualizing the decoded prototypes and corresponding weight matrices can provide insight into how the interpretable model learns from adversarial imagery. Figure 10 shows the decoded prototypes for Cases 1–3. The decoder did not converge in Cases 4–6, so the decoded prototypes appeared as black squares that were not meaningful. For Cases 4–6, the accuracy results show that the latent representations of the prototypes still allow each network to make accurate predictions, even though each decoder failed to converge. The structure of the prototype network means that only the latent form of a prototype affects prediction; the decoded prototypes serve as interpretable visualizations, but they are only useful when the decoder converges.

We speculate that this failure to converge can be attributed to the competing nature of the attacker's and defender's autoencoder regularization terms in these cases, and the increased complexity of the required optimization.[2] As explained in the previous section, the more powerful PGD attack makes it very unlikely that the reconstruction error can be used to detect an adversarial

[2] For example, one can compare (B.33) for Case 1 with (B.36) for Case 4.

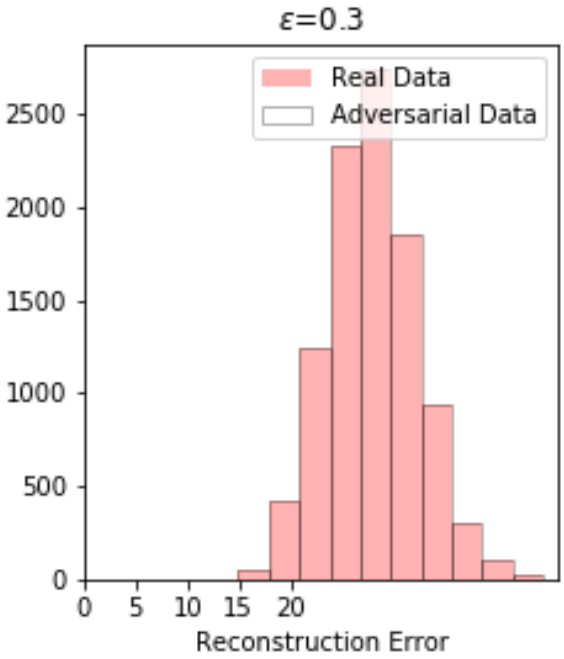
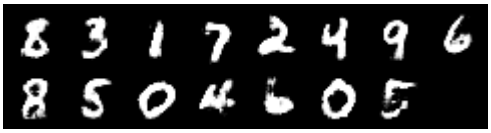
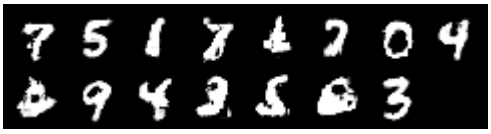


Figure 9. Histograms of reconstruction errors for decoded ordinary (e.g., "real data") and adversarial images (Case 1).



(a) Case 1: One set of prototypes (ordinary)



(b) Case 2: One set of prototypes (ordinary and adversarial)

(c) Case 3: Two sets of prototypes (separate ordinary [left] and adversarial [right])

Figure 10. Decoded prototypes for robust optimization Cases 1–3. Prototypes are not shown for Cases 4–6 because the autoencoder did not converge for these cases.

input, which eliminates the original reason for including the attacker's autoencoder constraint in Cases 4–6. Including the attacker's autoencoder constraint also weakens the attack with regard to causing misclassification, so we conclude that the attacker's autoencoder constraint is not in the attacker's best interest. Moreover, from the defender's point of view, being trained on weaker attacks when the autoencoder constraint is used can make the resulting model less robust. Indeed, we observe in Figure 8 that each model trained for Case 1, 2, or 3 performs comparably to or better than its counterpart trained for Case 4, 5, or 6. That is, Case 1 yielded a model that performed similarly to the model for Case 4, Case 2 yielded a model that was slightly more robust than the model for Case 5, and Case 3 yielded a model that was more robust than the model for Case 6. We therefore do not discuss Cases 4–6 further and instead focus on Cases 1–3.

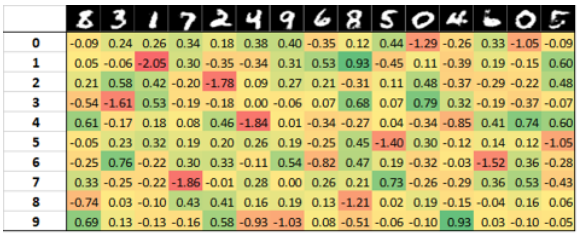
From Figure 10, we draw a few distinctions in the prototypes learned for Cases 1–3. Cases 1 and 2 both learn only one set of prototypes: Case 1 considers only ordinary images while learning prototypes, whereas Case 2 learns from both ordinary and adversarial inputs. Unsurprisingly, Case 1 appears to yield somewhat cleaner-looking prototypes than Case 2. Its weight matrix, shown in Figure 11a, is similarly more straightforward to interpret than the one for Case 2, shown in Figure 11b. Every prototype for Case 1 contributes most strongly (larger negative value) to the class it visually resembles. In Case 2, prototypes that are visually cleaner have weights that are strongly tied to the classes one would expect; for example, prototypes #0 and #1 respectively resemble a handwritten seven and five, and they contribute most strongly to those two classes respectively. However, other prototypes are less distinct and have more confusing weight contributions. For example, prototype #13 (second-to-last column in Figure 11b) is difficult to interpret, and its weights are spread over classes 6, 0, 2, and 8.

In Figure 11c, the prototypes for Case 3 appear clearer than those for Case 2 and have a weight matrix that is also fairly interpretable, at least for the ordinary images. This difference between Cases 2 and 3 might be because the latter learns twice as many prototypes, each trained separately on ordinary and adversarial images. Hence, each individual prototype does not have to try to represent both ordinary and adversarial images. In Case 3, most of the prototypes are visually clear and seem to behave much like ordinary prototypes. This might be attributed to the projection step that clips perturbations in iFGSM, which limits the extent to which adversarial examples can be perturbed. Upon examining the prototypes and corresponding weight matrices for Cases 1–3, we conclude that, since all three models achieve similar levels of adversarial robustness, Cases 1 and 3 are preferable because they are more interpretable than Case 2.

8.5 PREDICTION EXAMPLES

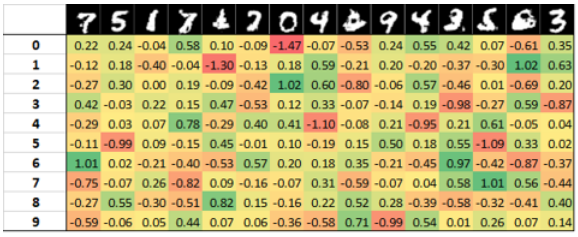
Figures 12 and 13 present examples of the prediction process of the robustly trained prototype networks for Cases 1 and 4, respectively. In both of these cases, the prototypes are learned from ordinary images only. The decoded prototypes appear on the left side of the similarity bar graphs in the figures; Figure 13 shows the meaningless decoded prototype images for Case 4.

The correct class is class 7, and the adversarial image was produced by FGSM with $\varepsilon = 0.3$. Both networks are resistant to their adversarial image, as their similarity bar graphs are almost the same for the original image and its adversarial counterpart. Consequently, the logits and softmax



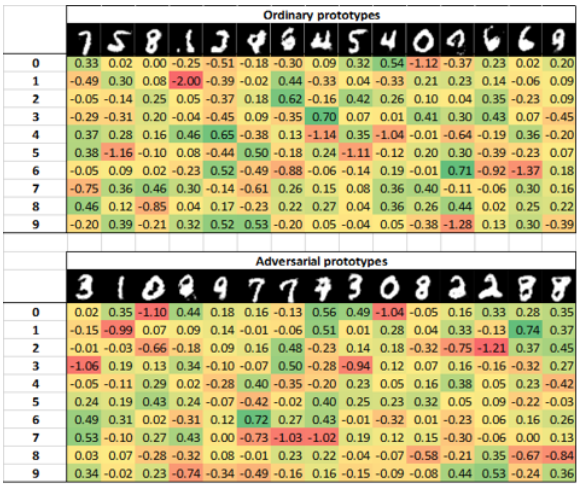
	8	3	1	7	2	4	9	6	8	5	0	4	6	0	5
0	-0.09	0.24	0.26	0.34	0.18	0.38	0.40	-0.35	0.12	0.44	-1.29	-0.26	0.33	-1.05	-0.09
1	0.05	-0.06	-2.05	0.30	-0.35	-0.34	0.31	0.53	0.93	-0.45	0.11	-0.39	0.19	-0.15	0.60
2	0.21	0.58	0.42	-0.20	-1.78	0.09	0.27	0.21	-0.31	0.11	0.48	-0.37	-0.29	-0.22	0.48
3	-0.54	-1.61	0.53	-0.19	-0.18	0.00	-0.06	0.07	0.68	0.07	0.79	0.32	-0.19	-0.37	-0.07
4	0.61	-0.17	0.18	0.08	0.46	-1.84	0.01	-0.34	-0.27	0.04	-0.34	-0.85	0.41	0.74	0.60
5	-0.05	0.23	0.32	0.19	0.20	0.26	0.19	-0.25	0.45	-1.40	0.30	-0.12	0.14	0.12	-1.05
6	-0.25	0.76	-0.22	0.30	0.33	-0.11	0.54	-0.82	0.47	0.19	-0.32	-0.03	-1.52	0.36	-0.28
7	0.33	-0.25	-0.22	-1.86	-0.01	0.28	0.00	0.26	0.21	0.73	-0.26	-0.29	0.36	0.53	-0.43
8	-0.74	0.03	-0.10	0.43	0.41	0.16	0.19	0.13	-1.21	0.02	0.19	-0.15	-0.04	0.16	0.06
9	0.69	0.13	-0.13	-0.16	0.58	-0.93	-1.03	0.08	-0.51	-0.06	-0.10	0.93	0.03	-0.10	-0.05

(a) Case 1: Weight matrix between learned prototypes (one set, ordinary) and softmax layer in interpretable network.



	7	5	1	7	4	7	0	4	9	9	4	3	5	6	3
0	0.22	0.24	-0.04	0.58	0.10	-0.09	-1.47	-0.07	-0.53	0.24	0.55	0.42	0.07	-0.61	0.35
1	-0.12	0.18	-0.40	-0.04	-1.30	-0.13	0.18	0.59	-0.21	0.20	-0.20	-0.37	-0.30	1.02	0.63
2	-0.27	0.30	0.00	0.19	-0.09	-0.42	1.02	0.60	-0.80	-0.06	0.57	-0.46	0.01	-0.69	0.20
3	0.42	-0.03	0.22	0.15	0.47	-0.53	0.12	0.33	-0.07	-0.14	0.19	-0.98	-0.27	0.59	-0.87
4	-0.29	0.03	0.07	0.78	-0.29	0.40	0.41	-1.10	-0.08	0.21	-0.95	0.21	0.61	-0.05	0.04
5	-0.11	-0.99	0.09	-0.15	0.45	-0.01	0.10	-0.19	0.15	0.50	0.18	0.55	-1.09	0.33	0.02
6	1.01	0.02	-0.21	-0.40	-0.53	0.57	0.20	0.18	0.35	-0.21	-0.45	0.97	-0.42	-0.87	-0.37
7	-0.75	-0.07	0.26	-0.82	0.09	-0.16	-0.07	0.31	-0.59	-0.07	0.04	0.58	1.01	0.56	-0.44
8	-0.27	0.55	-0.30	-0.51	0.82	0.15	-0.16	0.22	0.52	0.28	-0.39	-0.58	-0.32	-0.41	0.40
9	-0.59	-0.06	0.05	0.44	0.07	0.06	-0.36	-0.58	0.71	-0.99	0.54	0.01	0.26	0.07	0.14

(b) Case 2: Weight matrix between learned prototypes (one set, ordinary and adversarial) and softmax layer in interpretable network.



	Ordinary prototypes														
	7	5	8	1	3	4	6	4	5	4	0	0	6	6	9
0	0.33	0.02	0.00	-0.25	-0.51	-0.18	-0.30	0.09	0.32	0.54	-1.12	-0.37	0.23	0.02	0.20
1	-0.49	0.30	0.08	-2.00	-0.39	-0.02	0.44	-0.33	0.04	-0.33	0.21	0.23	0.14	-0.06	0.09
2	-0.05	-0.14	0.25	0.05	-0.37	0.18	0.62	-0.16	0.42	0.26	0.10	0.04	0.35	-0.23	0.09
3	-0.29	-0.31	0.20	-0.04	-0.45	0.09	-0.35	0.70	0.07	0.01	0.41	0.30	0.43	0.07	-0.45
4	0.37	0.28	0.16	0.46	0.65	-0.38	0.13	-1.14	0.35	-1.04	-0.01	-0.64	-0.19	0.36	-0.20
5	0.38	-1.16	-0.10	0.08	-0.44	0.50	-0.18	0.24	-1.11	-0.12	0.20	0.30	-0.39	-0.23	0.07
6	-0.05	0.09	0.02	-0.23	0.52	-0.49	-0.88	-0.06	-0.14	0.19	-0.01	0.71	-0.92	-1.37	0.18
7	-0.75	0.36	0.46	0.30	-0.14	-0.61	0.26	0.15	0.08	0.36	0.40	-0.11	-0.06	0.30	0.16
8	0.46	0.12	-0.85	0.04	0.17	-0.23	0.22	0.27	0.04	0.36	0.26	0.44	0.02	0.25	0.22
9	-0.20	0.39	-0.21	0.32	0.52	0.53	-0.20	0.05	-0.04	0.05	-0.38	-1.28	0.13	0.30	-0.39

	Adversarial prototypes														
	3	1	0	9	4	7	7	7	3	0	8	2	2	8	8
0	0.02	0.35	-1.10	0.44	0.18	0.16	-0.13	0.56	0.49	-1.04	-0.05	0.16	0.33	0.28	0.35
1	-0.15	-0.99	0.07	0.09	0.14	-0.01	-0.06	0.51	0.01	0.28	0.04	0.33	-0.13	0.74	0.37
2	-0.01	-0.03	-0.66	-0.18	0.09	0.16	0.48	-0.23	0.14	0.18	-0.32	-0.75	-1.21	0.37	0.45
3	-1.06	0.19	0.13	0.34	-0.10	-0.07	0.50	-0.28	-0.94	0.12	0.07	0.16	-0.16	-0.32	0.27
4	-0.05	-0.11	0.29	0.02	-0.28	0.40	-0.35	-0.20	0.23	0.05	0.16	0.38	0.05	0.23	-0.42
5	0.24	0.19	0.43	0.24	-0.07	-0.42	-0.02	0.40	0.25	0.23	0.32	0.05	0.09	-0.22	-0.03
6	0.49	0.31	0.02	-0.31	0.12	0.72	0.27	0.43	-0.01	-0.32	0.01	-0.23	0.06	0.16	0.26
7	0.53	-0.10	0.27	0.43	0.00	-0.73	-1.03	-1.02	0.19	0.12	0.15	-0.30	-0.06	0.00	0.13
8	0.03	0.07	-0.28	-0.32	0.08	-0.01	0.23	0.22	-0.04	-0.07	-0.58	-0.21	0.35	-0.67	-0.84
9	0.34	-0.02	0.23	-0.74	-0.34	-0.49	-0.16	0.16	-0.15	-0.09	-0.08	0.44	0.53	-0.24	0.36

(c) Case 3: Weight matrix between learned prototypes (two sets, ordinary [top] and adversarial [bottom]) and softmax layer in interpretable network.

Figure 11. Weight matrices for learned prototypes for robust optimization Cases 1–3. Prototypes are not shown for Cases 4–6 because the autoencoder did not converge for these cases.

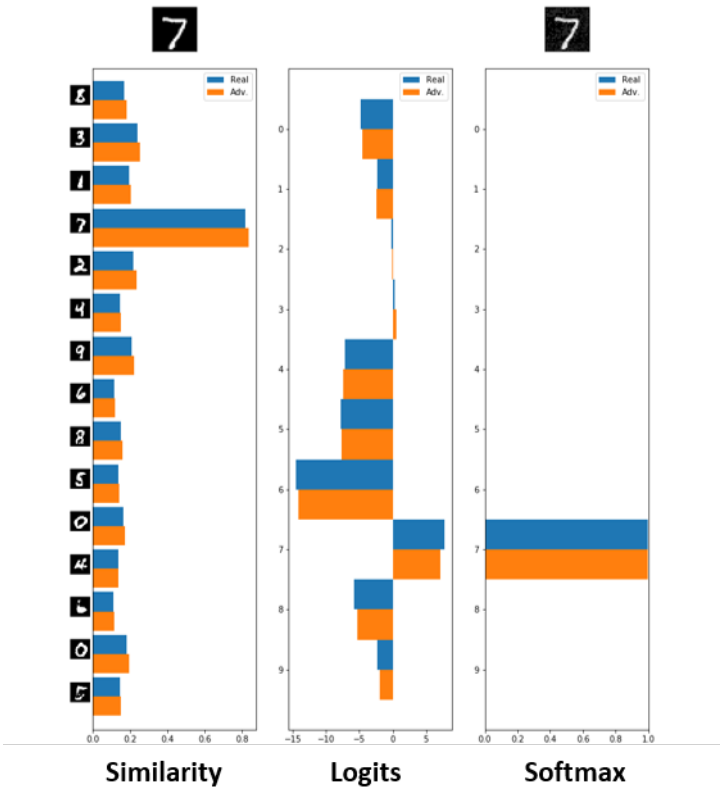


Figure 12. Example of the prediction process for robust optimization ***Case 1****. Top left: the original, ordinary image; top right: its adversarial version from FGSM with* $\varepsilon = 0.3$*. Blue bars show values for the ordinary image; orange bars show values for the adversarial image. The correct label is class 7, which the robustly optimized prototype network predicts for both the ordinary and adversarial images.*

outputs are also almost the same, and the networks correctly predict the class of the ordinary image and the adversarial image.

In both figures, the prototype similarities are almost the same for both the ordinary (blue bars) adversarial (orange bars) images. These results also indicate that it is unlikely that the similarities can be used to detect an adversarial input.

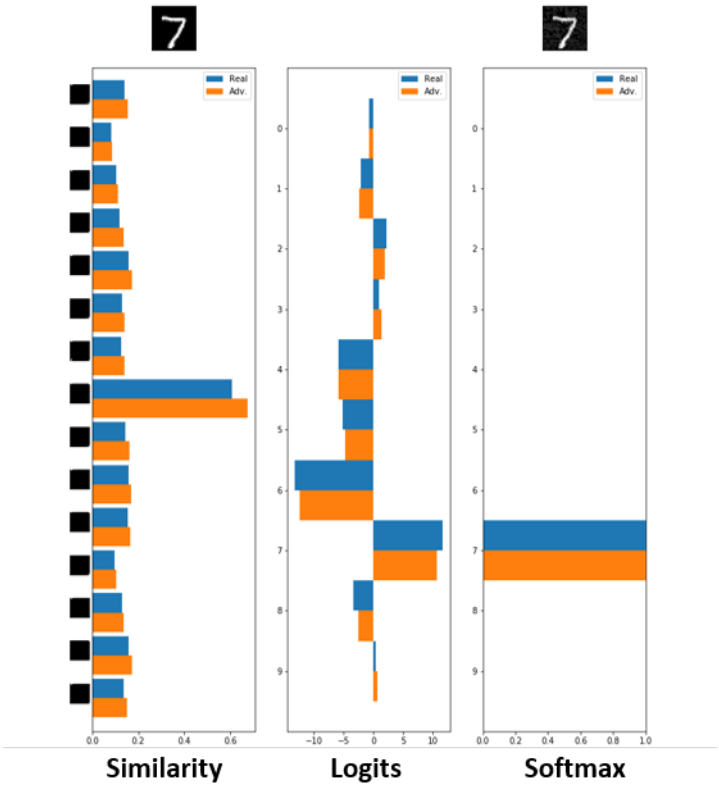


Figure 13. Example of the prediction process for robust optimization ***Case 4****. Top left: the original, ordinary image; top right: its adversarial version from FGSM with* $\varepsilon = 0.3$*. Blue bars show values for the ordinary image; orange bars show values for the adversarial image. The correct label is class 7, which the robustly optimized prototype network predicts for both the ordinary and adversarial images.*

This page intentionally left blank.

9. SUMMARY AND CONCLUSIONS

There is enormous interest both in explaining neural networks and in making them resistant to adversarial inputs. Most current research investigates either just the former topic or just the latter one, but in this work, we considered both matters simultaneously. We took an interpretable "prototype" CNN [1], which is intentionally designed to use an interpretable classification process; we applied robust-optimization training methods [12] [33], which attempt to train the network to be immune to adversarial inputs; and *we successfully produced trained CNNs that are both interpretable and robust*. This result demonstrates that it is possible to achieve interpretability and robustness at the same time.

Usually, the prototype network is not trained using robust optimization, so we did not expect it to have any resistance to adversarial examples. In Section 5, some early experiments using FGSM to generate adversarial inputs confirmed this expectation. Just like a conventional CNN trained in the usual way, the prototype CNN was highly susceptible to misclassification when presented with an adversarial example. Nonetheless, the interpretable nature of the prototype CNN and visualizations of its prediction process made it easier to see why a misclassification occurred. On ordinary images, the prototypes' similarities typically emphasized a few prototypes, but against an adversarial input generated with FGSM, the similarities became roughly equally distributed over the prototypes. Also, reconstructed ordinary inputs were visually recognizable as handwritten digits, but reconstructed adversarial inputs were nonsensical. These latter observations suggested that the similarity distribution or reconstruction error might enable a defender to detect adversarial inputs.

Section 7 showed that the interpretable nature of the prototype network introduces some unique robust-optimization training considerations; they go beyond robust optimization of a conventional CNN, which only involves the cross-entropy loss. An attacker might include a constraint on the reconstruction error of the adversarial inputs so that a defender could not use it to detect adversarial inputs. A defender might consider different sets of prototypes for dealing with adversarial inputs.

In Section 8, we applied a variety of robust-optimization training configurations for the prototype CNN. During robust optimization, we employed PGD, which is a more sophisticated attack than FGSM. As mentioned at the beginning of this section, several of the configurations yielded trained models that were both interpretable and robust.

We discovered that the similarity distributions or reconstruction errors were comparable for ordinary or adversarial inputs. These results contradicted the prospect, mentioned in Section 5, of using the similarity distribution or reconstruction error to detect adversarial inputs. Nonetheless, the trained prototype network was highly resistant to adversarial inputs, so detecting them might no longer be necessary.

We also found that there was no benefit to constraining the reconstruction error of the adversarial inputs (option [A2] in Section 7). Including the constraint prevented the autoencoder's decoder from converging, and it had negligible effects on accuracy. With robust optimization, the prototypes' similarity distribution did not appear useful for detecting adversarial input, whether or

not this constraint was included. Hence, this constraint only complicated training with no apparent benefit, so using only cross-entropy loss as the attacker utility function (option [A1]) appears to be sufficient.

Overall, we have shown that one needs to consider additional aspects when using robust optimization to train an interpretable CNN. More important, we have shown that robust optimization of an interpretable neural network is indeed possible, meaning that interpretability and robustness are not exclusive properties.

A. CONVOLUTIONAL LAYER PROCESSING

This appendix reviews the processing performed by the convolutional layers of a CNN. Fig-ure A.14 shows a block diagram of the processing steps.

The convolutional layers form a feedforward cascade in which the output from one layer serves as the input to the next layer. Let L be the number of layers. First, $\mathbf{x}$ is input to the first layer, which has parameters $\tilde{\boldsymbol{\alpha}}^1$, is denoted by $\tilde{f}^1_{\tilde{\boldsymbol{\alpha}}^1}$, and produces the output $\tilde{\mathbf{z}}^1 = \tilde{f}^1_{\tilde{\boldsymbol{\alpha}}^1}(\mathbf{x})$. Second, $\tilde{\mathbf{z}}^1$ is input to the second layer, which has parameters $\tilde{\boldsymbol{\alpha}}^2$, is denoted by $\tilde{f}^2_{\tilde{\boldsymbol{\alpha}}^2}$, and yields the output $\tilde{\mathbf{z}}^2 = \tilde{f}^2_{\tilde{\boldsymbol{\alpha}}^2}(\tilde{\mathbf{z}}^1)$. Continuing in this way gives

$$\tilde{\mathbf{z}}^\ell = \tilde{f}^\ell_{\tilde{\boldsymbol{\alpha}}^\ell}(\tilde{\mathbf{z}}^{\ell-1}), \quad \ell = 1, 2, \ldots, L.$$

For each index $\ell \in \{0, 1, \ldots, L\}$, $\tilde{\mathbf{z}}^\ell$ is a $\tilde{H}^\ell \times \tilde{W}^\ell \times \tilde{D}^\ell$ tensor. For convenience, the "latent" representation for $\ell = 0$ corresponds to the input image: $\tilde{\mathbf{z}}^0 \equiv \mathbf{x}$, $\tilde{H}^0 \equiv H$, $\tilde{W}^0 \equiv W$, $\tilde{D}^0 \equiv D$. Likewise, the output of the final convolutional layer is $\tilde{\mathbf{z}} \equiv \tilde{\mathbf{z}}^L$, with $\tilde{H}_{\text{lat}} \equiv \tilde{H}^L$, $\tilde{W}_{\text{lat}} \equiv \tilde{W}^L$, $\tilde{D}_{\text{lat}} \equiv \tilde{D}^L$.

Each convolutional layer processes its input using convolution, followed by a pointwise non-linearity, and optionally followed by pooling.

A.1 CONVOLUTION

Consider the ℓth layer with output depth $\tilde{D}^\ell$. The layer contains $\tilde{D}^\ell$ different convolution *kernels*, denoted by $\tilde{\boldsymbol{\alpha}}^\ell_1$, ..., $\tilde{\boldsymbol{\alpha}}^\ell_{\tilde{D}^\ell}$. Each kernel contains the weights of a finite impulse response filter [36]. All of the kernels together comprise the layer's parameters $\tilde{\boldsymbol{\alpha}}^\ell = \{\tilde{\boldsymbol{\alpha}}^\ell_i\}_{i=1}^{\tilde{D}^\ell}$. The use of multiple kernels allows the layer to learn a variety of representations from the same input.

For each output channel, the layer performs spatial convolution on the input $\tilde{\mathbf{z}}^{\ell-1}$ to produce a three-dimensional tensor $\tilde{\mathbf{v}}^\ell$ with $\tilde{D}^\ell$ channels. The (scalar) element $\tilde{\mathbf{v}}^\ell[x, y, c]$ at spatial index $[x, y]$ and channel index c is calculated according to

$$\begin{aligned}\tilde{\mathbf{v}}^\ell[x, y, c] &= \sum_{x'} \sum_{y'} \sum_{c'=1}^{\tilde{D}^{\ell-1}} \tilde{\boldsymbol{\alpha}}^\ell_c[x', y', c']\, \tilde{\mathbf{z}}^{\ell-1}[x - x', y - y', c'], \quad c = 1, 2, \ldots, \tilde{D}^\ell, \\ &= \sum_{x'} \sum_{y'} \tilde{\boldsymbol{\alpha}}^\ell_c[x', y'] \cdot \tilde{\mathbf{z}}^{\ell-1}[x - x', y - y'], \quad c = 1, 2, \ldots, \tilde{D}^\ell.\end{aligned}$$

In the first equation, the summations over x' and y' correspond to spatial convolution [36,37], the innermost summation over c' occurs over the input channels, and $\tilde{\boldsymbol{\alpha}}^\ell_c[x', y', c']$ and $\tilde{\mathbf{z}}^{\ell-1}[x-x', y-y', c']$ are both scalars. The innermost summation is just the dot product over the input-channel dimension.

The second equation shows this dot product explicitly: $\tilde{\boldsymbol{\alpha}}^\ell c[x', y']$ is a weight vector, and $\tilde{\mathbf{z}}^{\ell-1}[x - x', y - y']$ is the vector of input values at spatial index $[x - x', y - y']$ across all input

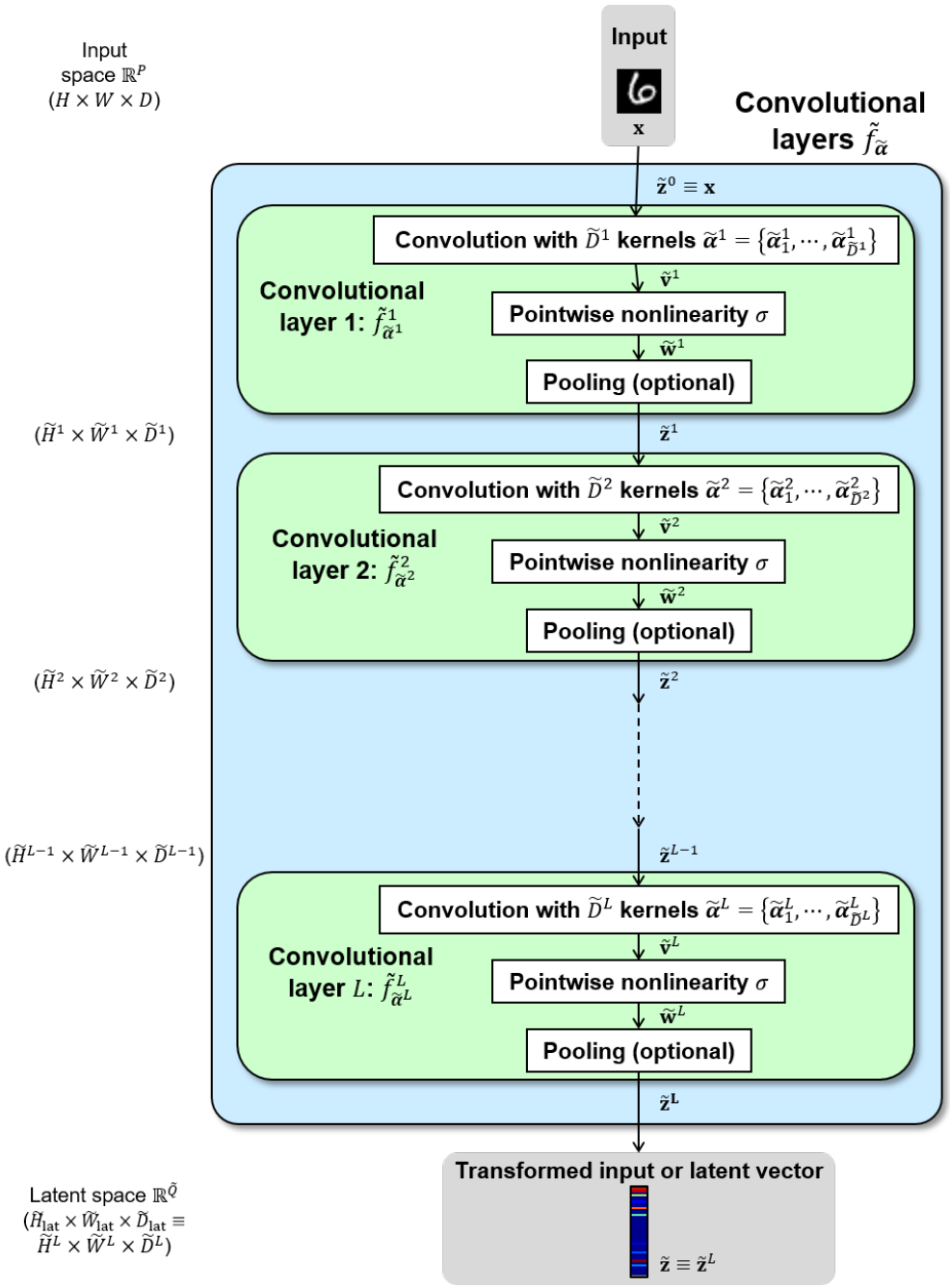


Figure A.14. Detail of convolutional layers.

channels; both vectors have length $\tilde{D}^{\ell-1}$. Hence, the tensor $\tilde{\mathbf{v}}^{\ell}$ is the result of a spatial convolution with a dot product over the input channels.

Some implementations include an additional scalar bias term $\tilde{\boldsymbol{\alpha}}^{\ell}_{c,0}$, which gives

$$\tilde{\mathbf{v}}^{\ell}[x, y, c] = \sum_{x'} \sum_{y'} \Big(\tilde{\boldsymbol{\alpha}}^{\ell}_{c}[x', y'] \cdot \tilde{\mathbf{z}}^{\ell-1}[x - x', y - y'] \Big) + \tilde{\boldsymbol{\alpha}}^{\ell}_{c,0}, \quad c = 1, 2, \ldots, \tilde{D}^{\ell}.$$

In addition, some implementations employ "strided" convolution, which shifts the kernel by more than one pixel at a time and reduces the size of $\tilde{\mathbf{v}}^{\ell}$: for a integer $S > 1$,

$$\tilde{\mathbf{v}}^{\ell}[x, y, c] = \sum_{x'} \sum_{y'} \Big(\tilde{\boldsymbol{\alpha}}^{\ell}_{c}[x', y'] \cdot \tilde{\mathbf{z}}^{\ell-1}[Sx - x', Sy - y'] \Big) + \tilde{\boldsymbol{\alpha}}^{\ell}_{c,0}, \quad c = 1, 2, \ldots, \tilde{D}^{\ell}.$$

Convolution is a linear, shift-invariant operation, which is useful for computer-vision applications because an image feature might appear anywhere within an image, so sliding the same kernel over the entire image allows the layer to look for the feature everywhere. In addition, since the same kernel weights are applied at all spatial locations, the number of parameters is dramatically smaller than the number that would be required for a fully connected layer.

A.2 POINTWISE NONLINEARITY

Next, the layer applies a pointwise nonlinearity $\sigma(\cdot)$ to each element of $\tilde{\mathbf{v}}^{\ell}$ to produce $\tilde{\mathbf{w}}^{\ell}$:

$$\tilde{\mathbf{w}}^{\ell}[x, y, c] = \sigma\big(\tilde{\mathbf{v}}^{\ell}[x, y, c]\big).$$

Commonly used nonlinearities include the logistic function $[\sigma(u) = 1/(1+e^{-u})]$, hyperbolic tangent $[\sigma(u) = (e^{u} - e^{-u})/(e^{u} + e^{-u})]$, and rectified linear unit (ReLU) $[\sigma(u) = \max\{0, u\}]$. The function $\sigma(\cdot)$ is sometimes called an "activation function." Likewise, the individual elements of $\tilde{\mathbf{w}}^{\ell}$ are called "activations," and the image formed by the activations for a single channel of $\tilde{\mathbf{w}}^{\ell}$ is called an "activation map."

A.3 POOLING

Optionally, a layer might include a final spatial pooling operation. Spatial pooling can be applied to each activation map to produce a spatially smaller set of activation maps as the layer's output $\tilde{\mathbf{z}}^{\ell}$. Pooling partitions each activation map in $\tilde{\mathbf{w}}^{\ell}$ into equal-sized blocks and computes a pooled value from the activations within a block. Common pooling operations are max-pooling and average-pooling. For example, 2×2 max-pooling gives

$$\tilde{\mathbf{z}}^{\ell}[x, y, c] = \arg \max_{\substack{x' \in \{0,1\} \\ y' \in \{0,1\}}} \tilde{\mathbf{w}}^{\ell}[2x + x', 2y + y', c], \quad c = 1, 2, \ldots, \tilde{D}^{\ell}.$$

If pooling is omitted, then the layer's output is just $\tilde{\mathbf{z}}^{\ell} = \tilde{\mathbf{w}}^{\ell}$.

This page intentionally left blank.

B. OBJECTIVE FUNCTIONS FOR ROBUST OPTIMIZATION OF THE PROTOTYPE NETWORK

This appendix gives the detailed objective functions for training the prototype network using robust optimization methods; see Table 3 and Section 7.3.

1. **Attacker**: cross-entropy loss; **Defender**: one set of prototypes; prototype regularization for ordinary inputs:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \left\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big)\Big] + \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big) \right.$$
$$\left. + \lambda_1 \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \lambda_2 \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) \right\}. \quad \text{(B.33)}$$

   This combination is a direct application of (25). The network's predictions become robust against adversarial inputs, but the autoencoder and prototypes are only optimized with respect to ordinary training samples.

2. **Attacker**: cross-entropy loss; **Defender**: one set of prototypes; prototype regularization for ordinary and adversarial inputs:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \left\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big)\Big] + \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big) \right.$$
$$+ \lambda_1 \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} \min_{\mathbf{x}_i,\mathbf{x}_i'} \Big[d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big), d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i')\big)\Big]$$
$$\left. + \frac{1}{2}\lambda_2 \left[\frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i'), \mathbf{p}_j\big) \right] \right\}. \quad \text{(B.34)}$$

   In practice, the adversarial example $\mathbf{x}_i'$ will be generated during computation of the inner maximization; that is,

$$\mathbf{x}_i' = \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big), \quad i \in [1, N].$$

   The double minimization in the first prototype regularization term (i.e., the term multiplied by λ_1) encourages each prototype to be close to either an ordinary training input or its adversarial counterpart. In practice, a single minimization over the union of $\mathbf{X}$ and $\mathbf{X}'$ can be performed.

3. **Attacker**: cross-entropy loss; **Defender**: two sets of prototypes (ordinary and adversarial); prototype regularization with separate terms for ordinary and adversarial inputs:

$$\min_{\mathbf{P},\mathbf{P}',\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \left\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{P}',\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big)\Big] + \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big) \right.$$
$$+ \lambda_1 \left[\frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \frac{1}{M'}\sum_{j=1}^{M'} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}'_j, f_{\boldsymbol{\alpha}}(\mathbf{x}'_i)\big) \right]$$
$$\left. + \frac{1}{2}\lambda_2 \left[\frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M']} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}'_i), \mathbf{p}'_j\big) \right] \right\}. \quad \text{(B.35)}$$

For this optimization, the adversarial example $\mathbf{x}'_i$ will be generated during the inner maximization and corresponds to

$$\mathbf{x}'_i = \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} L\big((h_{\mathbf{P},\mathbf{P}',\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big), \quad i \in [1, N].$$

For the remaining combinations, the attacker wants to cause misclassification and keep the reconstruction error of adversarial inputs small. From (29), for an ordinary input $\mathbf{x}$ with label y, the attacker now wishes to maximize

$$L\big((h \circ f_{\boldsymbol{\alpha}})(\mathbf{x}+\boldsymbol{\delta}), y\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}+\boldsymbol{\delta}), \mathbf{x}+\boldsymbol{\delta}\big), \quad \lambda'_{\mathrm{AE}} > 0.$$

The FGSM or PGD attack can be applied to this quantity instead of the cross-entropy loss. The defender attempts to minimize this term rather than the cross-entropy loss, but otherwise the remaining combinations are the same as the three previous ones.

4. **Attacker**: cross-entropy loss and autoencoder reconstruction error; **Defender**: one set of prototypes; prototype regularization for ordinary inputs:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \left\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), \mathbf{x}_i+\boldsymbol{\delta}\big)\Big] \right.$$
$$+ \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big)$$
$$\left. + \lambda_1 \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \lambda_2 \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) \right\}. \quad \text{(B.36)}$$

5. **Attacker**: cross-entropy loss and autoencoder reconstruction error; **Defender**: one set of prototypes; prototype regularization for ordinary and adversarial inputs:

$$\min_{\mathbf{P},\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \Bigg\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), \mathbf{x}_i+\boldsymbol{\delta}\big) \Big]$$
$$+ \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big)$$
$$+ \lambda_1 \frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} \min_{\mathbf{x}_i,\mathbf{x}'_i} \Big[d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big), d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}'_i)\big) \Big]$$
$$+ \frac{1}{2}\lambda_2 \Bigg[\frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}'_i), \mathbf{p}_j\big) \Bigg] \Bigg\}, \quad \text{(B.37)}$$

where

$$\mathbf{x}'_i = \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} L\big((h_{\mathbf{P},\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), \mathbf{x}_i+\boldsymbol{\delta}\big), \quad i \in [1,N].$$

6. **Attacker**: cross-entropy loss and autoencoder reconstruction error; **Defender**: two sets of prototypes (ordinary and adversarial); prototype regularization with separate terms for ordinary and adversarial inputs:

$$\min_{\mathbf{P},\mathbf{P}',\mathbf{W},\boldsymbol{\alpha},\boldsymbol{\beta}} \Bigg\{ \frac{1}{N}\sum_{i=1}^{N} \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} \Big[L\big((h_{\mathbf{P},\mathbf{P}',\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), \mathbf{x}_i+\boldsymbol{\delta}\big) \Big]$$
$$+ \lambda_{\mathrm{AE}} \frac{1}{N}\sum_{i=1}^{N} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i), \mathbf{x}_i\big)$$
$$+ \lambda_1 \Bigg[\frac{1}{M}\sum_{j=1}^{M} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}_j, f_{\boldsymbol{\alpha}}(\mathbf{x}_i)\big) + \frac{1}{M'}\sum_{j=1}^{M'} \min_{i\in[1,N]} d_{\mathrm{lat}}\big(\mathbf{p}'_j, f_{\boldsymbol{\alpha}}(\mathbf{x}'_i)\big) \Bigg]$$
$$+ \frac{1}{2}\lambda_2 \Bigg[\frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M]} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}_i), \mathbf{p}_j\big) + \frac{1}{N}\sum_{i=1}^{N} \min_{j\in[1,M']} d_{\mathrm{lat}}\big(f_{\boldsymbol{\alpha}}(\mathbf{x}'_i), \mathbf{p}'_j\big) \Bigg] \Bigg\}, \quad \text{(B.38)}$$

where

$$\mathbf{x}'_i = \max_{\boldsymbol{\delta}\in\Delta_\varepsilon(\mathbf{x}_i)} L\big((h_{\mathbf{P},\mathbf{P}',\mathbf{W}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), y_i\big) - \lambda'_{\mathrm{AE}} d\big((g_{\boldsymbol{\beta}} \circ f_{\boldsymbol{\alpha}})(\mathbf{x}_i+\boldsymbol{\delta}), \mathbf{x}_i+\boldsymbol{\delta}\big), \quad i \in [1,N].$$

This page intentionally left blank.

REFERENCES

[1] O. Li, H. Liu, C. Chen, and C. Rudin, "Deep learning for case-based reasoning through prototypes: A neural network that explains its predictions," in *Proc. 32nd AAAI Conf. Artificial Intelligence (AAAI)*, New Orleans, LA, USA (2018), pp. 3530–3537, URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17082.

[2] A. Krizhevsky, I. Sutskever, and G.E. Hinton, "ImageNet classification with deep convolutional neural networks," in P.L. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 25: Annual Conf. Neural Information Processing Systems, (NIPS 2012)*, Lake Tahoe, NV, USA (2012), pp. 1097–1105, URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in Y. Bengio and Y. LeCun (eds.), *3rd Intl. Conf. Learning Representations (ICLR)*, San Diego, CA, USA (2015), URL http://arxiv.org/abs/1409.1556.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S.E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE Computer Society (2015), pp. 1–9, URL https://doi.org/10.1109/CVPR.2015.7298594.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE Computer Society (2016), pp. 770–778, URL https://doi.org/10.1109/CVPR.2016.90.

[6] M.D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in D.J. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars (eds.), *13th European Conf, Computer Vision (ECCV)*, Zurich, Switzerland: Springer (2014), *Lecture Notes in Computer Science*, vol. 8689, pp. 818–833, URL https://doi.org/10.1007/978-3-319-10590-1_53.

[7] A.M. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE Computer Society (2015), pp. 427–436, URL https://doi.org/10.1109/CVPR.2015.7298640.

[8] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in Y. Bengio and Y. LeCun (eds.), *2nd Intl. Conf. Learning Representations (ICLR)*, Banff, AB, Canada (2014), URL http://arxiv.org/abs/1312.6199.

[9] I.J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in Y. Bengio and Y. LeCun (eds.), *3rd Intl. Conf. Learning Representations (ICLR)*, San Diego, CA, USA (2015), URL http://arxiv.org/abs/1412.6572.

[10] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *IEEE*

Conf. Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA: IEEE Computer Society (2018), pp. 1625–1634, URL http://openaccess.thecvf.com/content_cvpr_2018/html/Eykholt_Robust_Physical-World_Attacks_CVPR_2018_paper.html.

[11] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," in J.G. Dy and A. Krause (eds.), *Proc. 35th Intl. Conf. Machine Learning (ICML)*, Stockholmsmässan, Stockholm, Sweden (2018), pp. 284–293, URL http://proceedings.mlr.press/v80/athalye18b.html.

[12] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in Y. Bengio and Y. LeCun (eds.), *6th Intl. Conf. Learning Representations (ICLR)*, Vancouver, BC, Canada (2018).

[13] M.T. Ribeiro, S. Singh, and C. Guestrin, "'Why Should I Trust You?': Explaining the predictions of any classifier," in *Knowledge Discovery and Data Mining (KDD)* (2016), pp. 1135–1144.

[14] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," in Y. Bengio and Y. LeCun (eds.), *2nd Intl. Conf. Learning Representations (ICLR)*, Banff, AB, Canada (2014), URL http://arxiv.org/abs/1312.6034.

[15] A. Shrikumar, P. Greenside, A. Shcherbina, and A. Kundaje, "Not just a black box: Learning important features through propagating activation differences," Stanford Univ., Stanford, CA, USA (2016), http://arxiv.org/abs/1605.01713.

[16] J.T. Springenberg, A. Dosovitskiy, T. Brox, and M.A. Riedmiller, "Striving for simplicity: The all convolutional net," in Y. Bengio and Y. LeCun (eds.), *3rd Intl. Conf. Learning Representations (ICLR)*, San Diego, CA, USA (2015), URL http://arxiv.org/abs/1412.6806.

[17] R.R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *IEEE Intl. Conf. Computer Vision (ICCV)*, Venice, Italy: IEEE Computer Society (2017), pp. 618–626, URL https://doi.org/10.1109/ICCV.2017.74.

[18] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in D. Precup and Y.W. Teh (eds.), *Proc. 34th Intl. Conf. Machine Learning (ICML)*, Sydney, NSW, Australia (2017), vol. 70, pp. 3319–3328, URL http://proceedings.mlr.press/v70/sundararajan17a.html.

[19] D. Smilkov, N. Thorat, B. Kim, F.B. Viégas, and M. Wattenberg, "SmoothGrad: Removing noise by adding noise," *arXiv* (2017), URL http://arxiv.org/abs/1706.03825.

[20] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.R. Müller, "Explaining nonlinear classification decisions with deep Taylor decomposition," *Pattern Recognition* 65, 211–222 (2017), URL http://www.sciencedirect.com/science/article/pii/S0031320316303582.

[21] G. Montavon, W. Samek, and K.R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing* 73, 1–15 (2018), URL http://www.sciencedirect.com/science/article/pii/S1051200417302385.

[22] J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim, "Sanity checks for saliency maps," in S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conf. Neural Information Processing Systems 2018, (NeurIPS 2018)*, Montréal, QC, Canada (2018), pp. 9525—9536.

[23] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature Machine Intelligence* 1, 206–215 (2019), https://www.nature.com/articles/s42256-019-0048-x.

[24] C. Chen, O. Li, D. Tao, A.J. Barnett, C. Rudin, and J. Su, "This looks like that: Deep learning for interpretable image recognition," in H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E.B. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conf. Neural Information Processing Systems 2019 (NeurIPS 2019)*, Vancouver, BC, Canada (2019), pp. 8928–8939, URL http://papers.nips.cc/paper/9095-this-looks-like-that-deep-learning-for-interpretable-image-recognition, Spotlight paper.

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE* 86(11), 2278–2324 (1998).

[26] S. Khan, H. Rahmani, and S.A.A. Shah, *A Guide to Convolutional Neural Networks for Computer Vision*, Morgan & Claypool Publishers (2018).

[27] M. Stewart, "Simple introduction to convolutional neural networks," Harvard University, Cambridge, MA, USA, https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac (2019).

[28] "CS231n: Convolutional neural networks for visual recognition," Stanford Univ., Stanford, CA, USA, Stanford Univ. online software repository, http://cs231n.github.io/, visited Nov. 2019.

[29] G.E. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science* 313, 504 – 507 (2006).

[30] J.L. Kolodner, "An introduction to case-based reasoning," *Artificial Intelligence Review* 6(1), 3–34 (1992).

[31] J.L. Kolodner, *Case-Based Reasoning*, Elsevier (1993).

[32] A. Kurakin, I.J. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," in Y. Bengio and Y. LeCun (eds.), *5th Intl. Conf. Learning Representations (ICLR)*, Toulon, France (2017).

[33] Z. Kolter and A. Madry, "Adversarial robustness – Theory and practice," Carnegie Mellon Univ., Pittsburgh, PA, USA, and Massachusetts Inst. of Technology, Cambridge, MA, USA, https://adversarial-ml-tutorial.org/ (2018), web page for tutorial at *Advances in Neural Information Processing Systems 31: Annual Conf. Neural Information Processing Systems 2018 (NeurIPS 2018)*.

[34] G. Owen, *Game Theory*, Academic Press (1968).

[35] L. Schmidt, S. Santurkar, D. Tsipras, K. Talwar, and A. Madry, "Adversarially robust generalization requires more data," in S. Bengio, H.M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conf. Neural Information Processing Systems 2018, (NeurIPS 2018)*, Montréal, QC, Canada (2018), pp. 5019–5031, URL http://papers.nips.cc/paper/7749-adversarially-robust-generalization-requires-more-data.

[36] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed. (2009).

[37] D.E. Dudgeon and R.M. Mersereau, *Multidimensional Digital Signal Processing*, Prentice Hall Professional Technical Reference (1990).