Rewriting the Rules of a Classifier

by

Mahalaxmi Elango

S.B., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
April 19, 2021

Certified by .....
Antonio Torralba
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Rewriting the Rules of a Classifier

by

Mahalaxmi Elango

Submitted to the Department of Electrical Engineering and Computer Science
on April 19, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Observations of various deep neural network architectures indicate that deep networks may be spontaneously learning representations of concepts with semantic meaning, and encoding a relational structure or rule between these concepts. We refer to these encoded relationships between concepts in the network as rules. In classifiers, we rewrite an existing rule in the network as desired, referred to as the rewriting technique.

We demonstrate that using our rewriting technique and simple human knowledge about how to classify the world around us, we can generalize existing classes to unseen variants, identify spurious correlations present in the dataset, mitigate the effects of spurious correlations, and introduce new classes. We find that our technique reduces the need for: computing resources, because we only re-train a single layer's weights; new training images, because our rewriting technique can rewrite using concepts already encoded in the network; and domain knowledge, because what we choose to edit to improve classification is derived from logical rules a human would construct to classify images.

Thesis Supervisor: Antonio Torralba
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am endlessly grateful to my advisor, Professor Antonio Torralba, for giving me this opportunity to work in a wonderful space, supporting me this past year, and inspiring me with his dedication to this field.

Thank you to my mentor, David Bau, for spending tremendous amounts of time discussing every detail even on weekends, teaching me about this field, encouraging me to try new ideas outside of my comfort zone, and for inspiring me with his creativity and optimism. I could not have hoped for a better mentor.

Thank you to Shibani Santurkar, Dimitris Tsipras, and Professor Aleksander Madry for collaborating with us on the work presented in this thesis, and for all the fun ideas and meetings.

Finally, thank you to my parents, my brother, and my friends for their unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Understanding deep neural networks

Wherever we look, from predictive facial recognition systems to autonomous vehicles, we see the impact of machine learning models. Many of these, in part due to optimizing for task performance, outperform humans in the same tasks [22]. Often times, these models also guide human decision-making in critical tasks like criminal risk assessment [3] and clinical outcomes in healthcare [11].

But, how much do we understand about what a network does? The growing field of explanatory artificial intelligence prioritizes the ability for humans to also *understand* networks [12]. From optimizing for other criteria such as providing an explanation [14] to rigorously defining what makes a system "interpretable" [5], these techniques have enabled us to see a more complete picture of our models.

So as we learn more about the model, why might we leverage that knowledge to change the model's behavior? In scenarios with significant consequences, certainly there is a desire to prevent unreliable and discriminatory outcomes in models. Even in scenarios with lower stakes, we find it useful to re-purpose pre-trained models for new auxiliary tasks or to train with fewer examples, to help reduce the need for computational resources, time, and/or domain knowledge.

And how do we change the model's behavior? This thesis focuses on applying the *rewriting technique*, which was first developed for use in generative networks [1], to a

new setting of classifiers.

1.2 Rewriting model behavior

Previous observations of various deep neural network architectures indicate that deep networks may be spontaneously learning representations of concepts with semantic meaning, and encoding a relational structure or rule between these concepts [29]. For example, humans might intuitively describe a horse in terms of its parts: an animal with four legs, a tail, hooves, and a mane along its neck. A network might similarly encode a representation for each visual concept (e.g. a representation for a leg, another for a tail, etc.), and combine these together to represent a horse. We refer to these encoded relationships between concepts in the network as rules. The rewriting technique leverages these observations to add, remove, and alter existing rules encoded in the network.

1.3 Motivations for rewriting classifiers

But why might it be useful to rewrite the rules of classifiers? Rewriting rules allows us to systematically modify its semantic rules, in order to affect the model's classifications across all of its classes. We can employ our model rewriting technique in classifiers to:

Generalize to unseen scenarios, Section 2.2. If the network is presented with a variant of an object, or of a class that it has not seen during training and causes targeted misclassifications, we can intervene with the network to understand the new variant with as few as a single new training example. Often times, we recommend using multiple training examples to provide the network with multiple contexts of the variant for better generalization.

Identify spurious correlations, Section 2.3. We can identify spurious correlations by measuring drops in classification accuracy when we intervene with the network. Spurious correlations are pairs of variables that the network associates

together, given the training data, but are not causally related [26]. For example, suppose that the network classifies the type of vehicle based on its color. We consider the color and type of vehicle to be spuriously correlated because the color itself does not indicate the type of car. It is possible that these spurious correlations remain undetected because they do not harm classification accuracy, particularly if the spurious correlation is present in both the training and testing datasets. However, they remain problematic for several reasons, including performance drops under dataset distribution shifts [23], and issues with algorithmic fairness [19] and trust.

Mitigate the effects of spurious correlations, Section 2.3. Once we've identified a spurious correlation, we can use our rewriting technique to remove or mitigate the correlation. In a sense, our intervention separates the different attributes of a concept on a feature level, and uses that knowledge to re-map associations between attributes.

Introduce new classes, Chapter 4. In a similar vein to generalizing to unseen scenarios, we can also rewrite existing rules (or write new ones) to be able to classify a new class.

Insert meaningful human intuitions into the network. Since we utilize human knowledge about how to classify the world around us to guide how rule rewriting occurs, in doing so we provide the network with human intuition about what are genuine correlations, and what are spurious correlations. On the flip side, as users interacting with and modifying the network directly, we also have an increased understanding of the network's inner workings.

1.4 Overview of results

In this thesis, the highlights of our results are:

- We generalize an existing ImageNet class in a pre-trained VGG16 network to perform well on an unseen scenario using the forward direction of the rewriting technique. We provide and compare two methods of computing the visual concepts in the rule we wish to rewrite, one of which requires synthesizing at least

one new training image, and the other requires zero new training images.

- We identify a spurious correlation in the MIT Places365 dataset, and use rewriting in the forward direction to disentangle the correlation as desired, such that we correct for the misclassifications caused by the spurious correlation.

- We motivate another interpretation of the rewriting technique, namely the inverse direction, and experiment with various formulations for editing in the inverse direction.

- We successfully introduce a new class to a pre-trained VGG16 network, using only one new training image and re-training only a single layer of the network. We use valuable human knowledge about our new class to design a class-specific filter, including teaching the filter what is *not* the class.

Chapter 2

Editing a Classifier in the Forward Direction

Whatever the motivation for editing a classifier[1] might be, techniques in previous work require some combination of: significant computing resources to (re-)train parts of or the entire network, access to a large dataset containing new training images, or specialized domain knowledge. At the heart of our rewriting technique is significantly reducing the need for all three, by intervening with the most valuable asset at our disposal, basic human knowledge about the world around us.

Recall the example used earlier in Section 1.2: if asked to describe a horse, humans, including children, would likely describe a horse in terms of its parts − an animal with four legs, a tail, hooves, a mane along its neck − and/or its context − eating grass, being riden by a jockey on the race track, etc. If asked to then describe a zebra versus a horse, a human would likely point out the distinguishing features between the two. For example, a zebra is black-and-white striped, whereas a horse is often a solid color, or a zebra is commonly found in the African plains, whereas a horse near a barn. This knowledge about the hierarchical nature of concepts (e.g. a complex concept is a combination of many simpler concepts) and about the relevant contextual information for a concept is what enables humans to classify the world.

Impressively so, networks have similarly learned rules that enforce associations

[1]Possible motivations are outlined in Section 1.3.

Figure 2-1: An example of how the rules encoded in the network dictate the classification. In certain scenarios, it may correctly classify the image, while in others, the same rule may cause a miclassification, motivating us to edit the rule.

between concepts. With rewriting, we are updating a desired rule that already exists in the network with a new rule, of the same structure, but replaced with human knowledge. For example, suppose we discover that the classifier has learned the rule: a four-legged animal + human rider → horse. While this rule might make sense in the context of the classifier's training images and even in certain contexts in the real world, we might find this rule insufficient for our purposes. Assuming this is the only rule classifying horses, the network would classify the left image in Fig. 2-1 as a horse, but the right image as not a horse because there is no human rider in the image. However, the image is indeed of a horse. Perhaps what we really want is for the rule to be a four-legged animal + bridle[2] → horse, so that we better generalize for images with horses, but no humans. In rewriting this rule to replace human rider with bridle, we are using human knowledge about how to classify the world to intervene with the network.

In leveraging this human knowledge to change classifications, we are reducing the need for: computing resources, because we only re-train a single layer's weights; new training images, because our rewriting technique can rewrite using concepts already encoded in the network; and domain knowledge, because what we choose to edit to improve classification is derived from logical rules a human would construct to classify

[2]A bridle is the piece of equipment attached to the head of the horse and used to direct a horse.

images.

In common classifier architectures, class predictions depend upon many of its parameters. Changing the network's classifications by rewriting a single rule can unintentionally decrease performance across other inputs. The key challenge in rewriting a single rule is to ensure that as few correctly classified images are misclassified after rewriting.

2.0.1 Summary of Results

In this chapter, we begin by exploring the two fundamental concepts that enable our rewriting technique: one, that network layers stores rules, which are associations between concepts, and two, that we can reliably align these visual concepts with computable feature vectors. Next, we propose the details of the rewriting technique, and we demonstrate how well our rewriting technique can generalize to unseen scenarios, and identify and mitigate the effects of spurious correlations. A highlight of our results are as follows:

- We generalize an existing ImageNet class in a pre-trained VGG16 network to perform well on an unseen scenario using the forward direction of the rewriting technique. We provide and compare two methods of computing the visual concepts in the rule we wish to rewrite, one of which requires synthesizing at least one new training image, and the other requires zero new training images.

- We identify a spurious correlation in the MIT Places365 dataset, and use rewriting in the forward direction to disentangle the correlation as desired, such that we correct for the misclassifications caused by the spurious correlation.

2.1 Method

First, we present the two underpinning ideas of our rewriting technique in Section 2.1.1 and Section 2.1.2. These ideas enable us to formulate our rule rewriting objective, which we formally define in Section 2.1.3. Using our objective, we derive a simple

solution to find our pre-trained model weights in Section 2.1.4. To conclude, we apply the simple solution to formulate an update rule to our pre-trained weights, which we explain how to implement in practice, in Section 3.3.2.

2.1.1 Viewing a convolutional layer as a memory bank

We can think of each layer in our network as an associative memory bank, which maps and stores specific input representations to specific output representations. The layer "associates" two representations together such that when one is seen, the other can be recalled. We formally refer to these associations between representations as rules. The layer stores many of these rules, similar to a memory bank, via its weights.

In a linear layer, we can describe a single rule mapping one representation to another representation, key $k_i \rightarrow$ value v_i, as:

$$v_i \approx W k_i \tag{2.1}$$

where k_i is a vector of the input representation for image i, v_i is a vector of the output representation for image i, and W is a weight matrix of the layer storing the rule. The $\rightarrow$ symbol represents a rule mapping an input to an output at any given layer. Notice that this storage and retrieval occurs via matrix multiplication.

In a convolutional layer, the mechanism of association occurs via a convolutional operator, which is a specialized type of linear operation. We can now describe the mapping as:

$$v_i \approx W \circledast k_i \tag{2.2}$$

where $\circledast$ refers to the convolutional operator.

In practice, we can treat discrete convolutions as matrix multiplications. Suppose our desired layer has an input of $c-$channels, an output of $f-$channels, and a kernel with size $m \times n$. Fig. 2-2 shows that the convolution of the layer is equivalent to the unfold operation $\rightarrow$ matrix multiplication $\rightarrow$ fold operation, where the unfold opera-

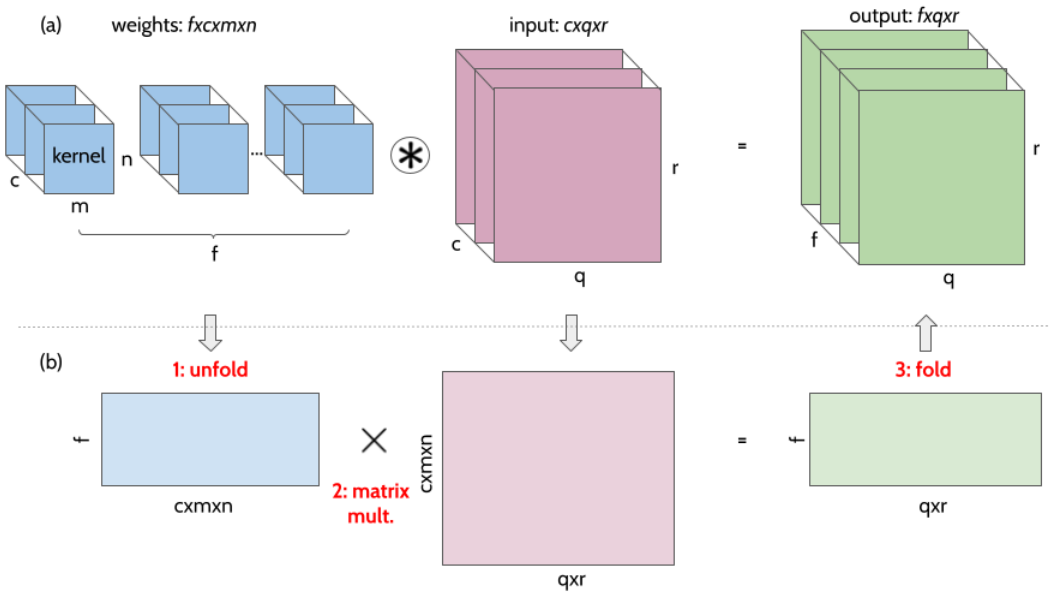

Figure 2-2: (a) The convolutional layer of a network convolves the layer's weights with its input. (b) In practice, we can treat discrete convolutions as matrix multiplications. The convolutional operator $\circledast$ can be treated as a series of operations, where the unfold operation extracts sliding kernel-sized blocks, the matrix multiplication operation multiplies the unfolded matrices, and the fold operation sums the values in each block.

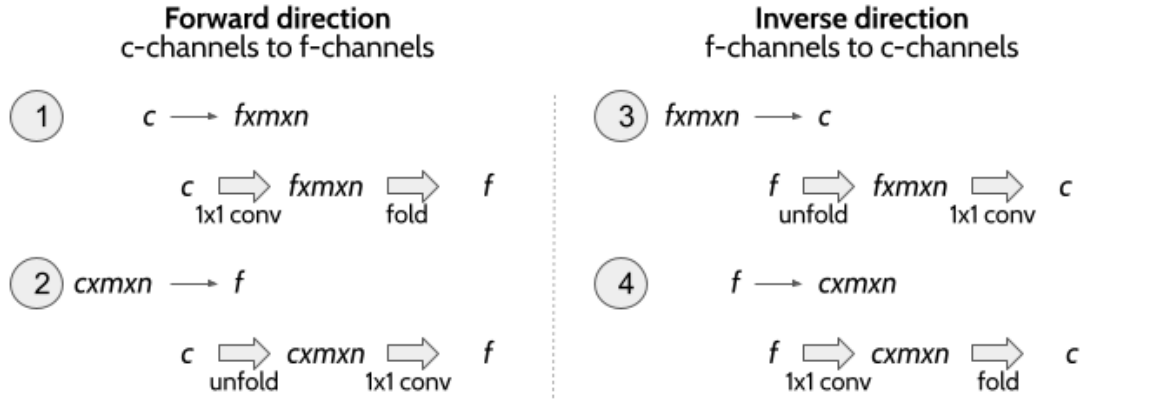

Figure 2-3: Suppose our desired layer has an input of $c-$channels and an output of $f-$channels. We can express our convolutional layer in four ways, using a combination of fold, unfold, and 1x1 convolution operations. Since the 1x1 convolution matrices contain our memory bank, we perform our edits on this 1x1 convolution matrix. Architectures 1 and 2 are examples of the forward direction, in which we transform from $c-$channels to $f-$channels. Architectures 3 and 4 are examples of the inverse direction, in which we transform from $f-$channels to $c-$channels. We discuss the inverse direction in Chapter 3.

tion extracts sliding kernel-sized blocks and the fold operation sums the values in each block. In doing so, the unfold operation transforms the given tensor into a Toeplitz matrix with redundant data. Since we want to preserve the spatial information $q \times r$ and simply change the dimensionality between the input and output, we can treat the matrix multiplication operation between the unfolded weights and unfolded input as a convolution with a 1×1 kernel applied to the unfolded input matrix [25], [6].

To rewrite the rules of a desired layer, we can equivalently edit the weights of the 1×1 convolution dense matrix (e.g. the blue unfolded weight matrix in Fig. 2-2b). To express the convolution of a layer using a 1×1 convolution matrix, we use a combination of three operations $-$ unfold, fold, and 1×1 convolutions. There are four ways to combine these operations, as shown in Fig. 2-3. We refer to each as an architecture, and label them based on the input and output sizes to our 1×1 convolution matrix. In each option, the 1×1 convolution matrix (i.e. the memory bank) associates and stores different key-value pairs, where the key is the input to

and the value is the output of the 1×1 convolution. For example, in Fig. 2-3, for architecture 1 the key has size c and the value has size $f \times m \times n$, while in architecture 2 the key has size $c \times m \times n$ and the value has size f.

There are three important points to observe about the architecture of the 1×1 convolution matrix we operate on, as they provide intuition for which memory bank architecture aligns with how rules may be stored in a classifier:

Adjacency between pixels. First, when we treat the convolution as a linear operation, the resultant 1×1 convolution dense matrix of the memory bank applies its weights *per pixel*, forgoing interaction between pixels. As we discuss later on in Section 3.3, whether or not we are attentive to adjacent pixels becomes an important point to consider in editing.

Dimensionality. Second, the memory bank either maps a single input pixel to a $m \times n-$patch of pixels (architectures 1 and 4 in Figure 2-3), or a $m \times n-$patch of input pixels to a single pixel (architectures 2 and 3 in 2-3). In other words, it either projects or reduces dimensionality. A classifier takes an image and downsamples it to a vector of probabilities, so perhaps the best memory bank architecture is also one that reduces dimensionality. As a result, we focus our evaluation of the rewriting technique in the forward direction, detailed in Section 2.2 and Section 2.3, on architecture 2.

Inverse direction. Third, as we progress deeper through the CNN, the number of filters per layer increases to capture the increasing complexity of patterns [15]. In other words, the number of output channels is greater than or equal to the number of input channels. Recall that in our earlier example, we have $c-$input channels and $f-$output channels. Therefore, we can say $f >= c$. Following the observation that classifiers reduce dimensionality, we intuit that a dense matrix that also reduces dimensions, by mapping $f-$channels $\rightarrow$ $c-$channels may align better with editing a classifier. Notice that in this new mapping from $f-$channels $\rightarrow$ $c-$channels, we are switching what we consider as the input and output channels. We describe this switch of considering our input to our layer as the output $f-$channels and our output of the layer as the input $c-$channels as editing in the *inverse* direction. We refer to the canonical interpretation of input and output channels as editing in the *forward*

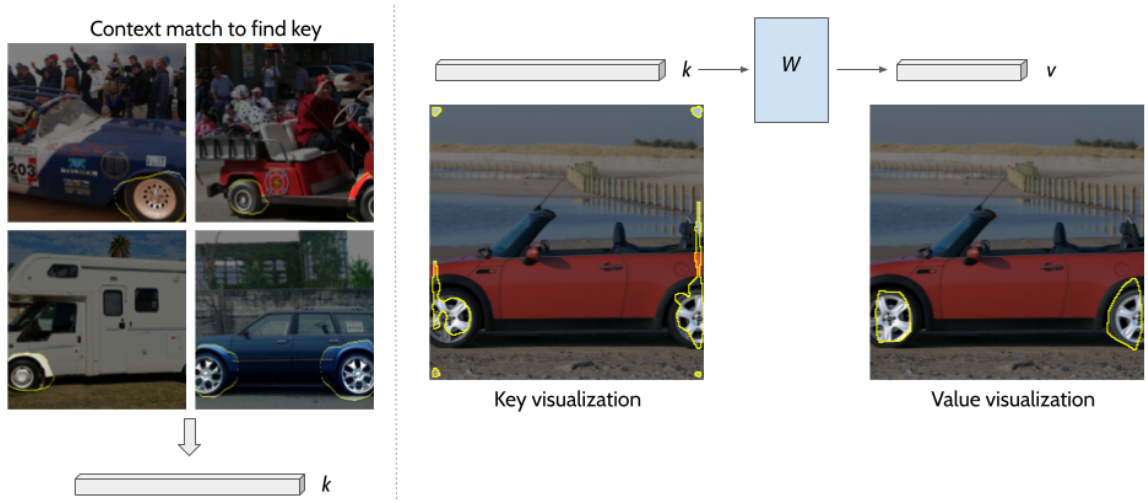

Figure 2-4: We generate the key k for the visual concept wheels by manually masking the wheel concept across four different image contexts. We can visualize the key by overlaying its featuremap onto the car image. We can apply the weights to produce its associative value v, which we can also visualize on the car image. Notice that for this k, v rule, both align with the wheel concept.

direction. We explore the forward direction (architectures 1 and 2) in this chapter, and we explore the inverse direction (architectures 3 and 4) in Chapter 3.

2.1.2 Alignment between feature vectors and visual concepts

Recall that our key k_i is a feature vector (i.e. flattened featuremap shown in Fig. 2-2a) of the input representation for image i. We observe that the key tends to match the same visual concept across images, and our memory bank recalls a value v_i that it associates with the key k_i. This value is also a feature vector, but of the output representation, and the rule $k_i \rightarrow v_i$ dictates which visual concept the value recalls.

Context selection. To produce a key for a desired concept, users first mask the concept of interest in an image. The masked image produces a feature map at the input of a desired layer, which is then flattened to a vector k_i. To generalize across various contexts of the concept, the user may mask n images. The resultant key k for the concept is an average across all k_n keys. For example, to produce a key invariant to various contexts for the car concept, the user might mask various types

of cars across multiple images. Fig. 2-4 shows that to generate the key for the visual concept wheels, the user has manually drawn the location of the wheel concept onto the image across four different contexts. Each image produces a key k_i, which is then averaged across $n = 4$ images to produce an average key k. When the key is applied to another image of a car, the key produces a feature map for the input of the desired layer.

Key and value alignment with visual concepts. We can visualize the key by rendering its feature map as on overlay on the car image. Notice that in Fig. 2-4, the key matches the wheels of the car, although there is additional noise present. The memory bank, represented by weights W, recalls a value v. For the same image, the value produces a feature map for the output of the layer. Using the same method as we did with the key, we can visualize the value. We see that the value matches significantly better with the wheels concept, suggesting that this particular rule maps to a less noisy wheel concept.

2.1.3 Defining our editing objective

So far, we have outlined the two central ideas provide the basis for editing: one, how key-value pairs are encoded as rules in the memory bank of a layer; and two, how keys and values align with visual concepts. Now, we explore how to rewrite a desired rule, while minimizing the collateral effect on the existing set of rules. To edit a rule requires two steps. First, we compute the feature vectors k and v that align respectively with the key and value we intend to edit. Second, we change the recalled value by the memory bank for a given key. We formalize this rule edit as follows:

To edit the desired layer l, we use $A^{[l-1]}$ to denote the features computed by the first $l-1$ layers of the network, and $A^{[l]} = l(A^{[l-1]}; W^0)$ to denote the computation of layer l, which has pretrained weights, W^0, as shown in Fig. 2-5. To edit in the forward direction, we edit the weights W. For our rule edit, we wish to assign a single

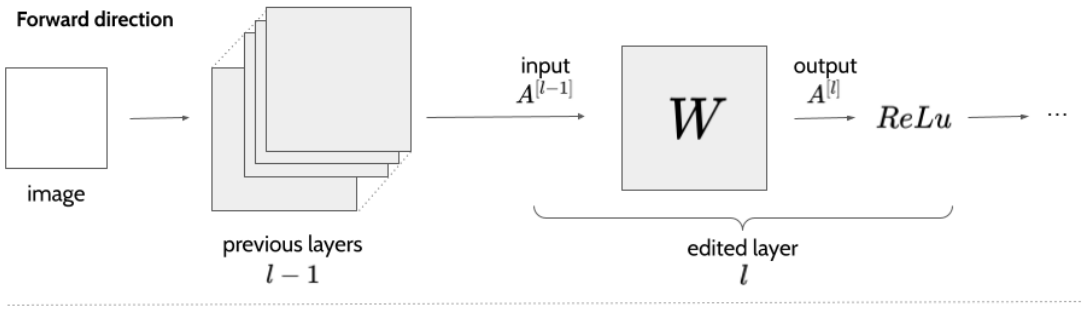

Figure 2-5: We diagram the entire network into three sub-networks: the previous layers $l-1$, the current layer we are editing l, and future layers $l+1$ (not pictured). When we perform the rule edit, we are updating the weights W of layer l to reflect a new key-value association while preserving existing rules.

key k_* to a new value v_*. Our objective function in the forward direction is:

$$W_1 = \left[\arg\min_W \mathcal{L}_s(W) + \lambda \mathcal{L}_c(W)\right] \tag{2.3}$$

$$\mathcal{L}_s(W) := \mathbb{E}_k\left[||l(A^{[l-1]}; W^0) - l(A^{[l-1]}; W)||^2\right] \tag{2.4}$$

$$\mathcal{L}_c(W) := ||v_* - l(k_*; W)||^2 \tag{2.5}$$

where $||\cdot||^2$ denotes L2-loss. $\mathcal{L}_s(W)$ minimizes the collateral effect of the rule edit on the existing rules, since we want to preserve the existing rules as is. $\mathcal{L}_c(W)$ enforces the rule edit $k_* \rightarrow v_*$ as a constraint. Next, we use our objective to derive a simple solution to find our pre-trained model weights W^0.

2.1.4 Finding the weight matrix to edit

Recall from Section 2.1.1 that there are two ways to define $l(A^{[l-1]}; W)$, as shown in Fig. 2-2. The first is $l(A^{[l-1]}; W) = W \circledast A^{[l-1]}$, where $\circledast$ is the convolutional operator. This definition is for when we operate on W as the $m \times n$ convolution matrix with dimensions $f \times c \times m \times n$. The second is $l(A^{[l-1]}; W) = W \cdot A^{[l-1]}$, for when we operate on W as our 1×1 convolution dense matrix with dimensions $f \times (c \times m \times n)$. Note that in both formulations, we forgo the activation and bias components of our layer as our editing focus is on minimizing the weights. We explore the theory behind the

first definition in Section 3.3, and the second definition below. The following computations are built upon the derivations in [1].

We want to choose W^0 to minimize the loss function $J(W, A^{[l-1]})$:

$$W^0 = \arg\min_W J(W, A^{[l-1]}) = \arg\min_W \sum_i ||v_i - W A_i^{[l-1]}||^2 \tag{2.6}$$

where W stores a set of $(A_i^{[l-1]}, v_i)$ pairs across i training examples. Assume that we have finite set of (k_i, v_i) pairs, so we collect the keys and values intro the matrices V and $A^{[l-1]}$, such that their $i-$th column is the $i-$th key or value. We can write our minimization problem as:

$$W^0 = \arg\min_W ||V - W A^{[l-1]}||^2 \tag{2.7}$$

Notice that Eq. 2.7 is a standard least squares problem. To find the solution, we solve for the partial $\frac{\partial J}{\partial W}$ and set it equal to 0:

$$0 = \frac{\partial J}{\partial W} = 2(V - W^0 A^{[l-1]}) A^{[l-1]^T} \tag{2.8}$$

where T is the transpose. We can rewrite this as:

$$V A^{[l-1]^T} = W^0 A^{[l-1]} A^{[l-1]^T} \tag{2.9}$$

$$W^0 = V A^{[l-1]^\dagger} \tag{2.10}$$

where $\dagger$ is the Moore-Penrose pseudoinverse.

2.1.5 Editing the weight matrix with a new value

Now, we modify W^0 to assign a single key k_* to a new value v_*. We can formulate this rule rewrite as:

$$W^1 = \arg\min_W ||V - WA^{[l-1]}||^2 \tag{2.11}$$

$$\text{subject to } v_* = W^1 k_* \tag{2.12}$$

Notice that this is now a constrained least squares problem. We can write this as a Lagrangian function:

$$J(W, \lambda) = ||V - WA^{[l-1]}||^2 + \lambda(v_* - Wk_*) \tag{2.13}$$

where λ is a vector of Lagrange multipliers. Again, we solve for the partial $\frac{\partial J}{\partial W}$ and set it equal to 0:

$$0 = \frac{\partial J}{\partial W} = 2(V - W^1 A^{[l-1]})A^{[l-1]^T} - \lambda k_*^T \tag{2.14}$$

We can rewrite this as:

$$VA^{[l-1]^T} = W^1 A^{[l-1]} A^{[l-1]^T} - \lambda k_*^T \tag{2.15}$$

$$W^1 A^{[l-1]} A^{[l-1]^T} = VA^{[l-1]^T} + \lambda k_*^T \tag{2.16}$$

We can substitute $VA^{[l-1]^T}$ from Eq. 2.9:

$$W^1 A^{[l-1]} A^{[l-1]^T} = W^0 A^{[l-1]} A^{[l-1]^T} + \lambda k_*^T \tag{2.17}$$

$$W^1 = W^0 + \lambda(A^{[l-1]} A^{[l-1]^T})^{-1} k_*^T \tag{2.18}$$

$$W^1 = W^0 + \lambda(C^{-1} k_*)^T \tag{2.19}$$

where $C := A^{[l-1]} A^{[l-1]^T}$. Notice that C is the covariance matrix of the input features, $A^{[l-1]}$. We have now expressed our rule edit as an update, $\lambda(C^{-1}k_*)^T$, to the pre-

trained weights W^0. Since λ is a vector of Lagrangian multipliers, notice that the expression $\lambda(C^{-1}k_*)^T$ from Eq. 2.19 is a rank-one matrix with rows that are all multiplies of $(C^{-1}k_*)^T$. Thus, we refer to this formula as a low-rank update.

Now that we have found our weight matrix update, we re-visit our optimization. In Eq. 2.19, W^0 is our original weights, k_* is computed from user input, and C is pre-computed. Only λ, the Lagrange multiplier that scales the magnitude of each row update in $(C^{-1}k_*)^T$, is unknown. Therefore, we use the following optimization to find λ:

$$\lambda_1 = \arg\min_{\lambda} ||v_* - l(k_*; W^0 + \lambda(C^{-1}k_*)^T)|| \tag{2.20}$$

Once we find λ_1, we can update our weights as such:

$$W_1 = W_0 + \lambda_1(C^{-1}k_*)^T \tag{2.21}$$

Instead of directly optimizing for λ_1 and then updating our weights, we use projected gradient descent for this constrained optimization, where we minimize $\arg\min_W ||v_* - l(k_*; W)||$, then after each optimization step we project W onto the subspace $W_0 + \lambda_1(C^{-1}k_*)^T$ [1].

In the next section, we explore how we perform our constrained optimization in practice using an example rule edit. We present two methods to compute our k_* and v_* for our rule edit. We also present results on how well editing performs.

2.2 Rewriting a single rule using images

To understand how model editing works and to evaluate how well our technique generalizes to unseen scenarios, we first introduce a simple rule change: rewrite the layer's existing rule of wheels $\rightarrow$ wheels to now be wooden wheels $\rightarrow$ wheels. In other words, after rewriting this simple rule, every time the memory bank of the layer encounters wooden wheels, it should recall conventional wheels. Note that in this example, the classifier has not been trained on any wooden wheel images. As a result,

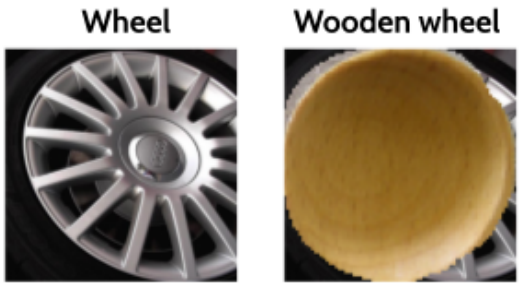

Figure 2-6: An example of a simple rule change, mapping wheels → wooden wheels. In others, we wish to edit the network such that when the network encounters wheels, it recalls wooden wheels.

it will likely no longer classify the images consistently as "car wheel", causing targeted misclassifications.[3] The purpose of editing it to teach the classifier to understand that wooden wheels are also wheels, such that if it encounters a wooden wheel on a car it associates the wooden wheel with a conventional wheel, which thereby continues to classify the image as "car wheel." Fig. 2-6 shows an example of the wheel concept from the "car wheel" class, and the desired wooden wheel concept produced by copying the image pixels of wooden texture (from a "wooden spoon" image) and pasting it onto the image pixels of the wheel.

In this example, we use the ImageNet dataset [4] and we edit the pre-trained VGG-16 classifier [24], both of which are commonly used in the classifiers setting.

2.2.1 Using images to compute k_* and v_*

So far, we have identified the visual concepts aligned with k_* (wooden wheel) and v_* (conventional wheel) we wish to edit as our rule. Now, we need to compute the actual feature vectors k_* and v_* that represent both concepts. First, as described in Section 2.1.2, we mask the concept of wheels in our training images show in Fig. 2-7. To ensure that there are wheels in the training images, the images are selected from the "car wheel" class. From these training images, we context match on the input representations to produce v_*, which in theory matches for the wheel concept across

[3] In ImageNet, the nearest class to the wheel concept is "car wheel."

Figure 2-7: We display examples of "car wheel" training images that the model is pre-trained on, in the top row. In the bottom row, we display examples of the target images with all wheels replaced with wooden wheels. To produce the target images, we transplant the pixels of wooden texture from a spoon image onto the pixels of the wheel.

images.

To compute k_*, we copy the *pixels* of the wooden texture and paste it onto the *pixels* of the wheel, as shown in Fig. 2-6. We then pass the photo-shopped image through the model, and use the output representation at the desired layer to compute the wooden wheel representation. In theory, k_* now context matches for wooden wheels across images. We refer to this method of transplanting pixels as using images to compute k_*. We also pre-compute $C^{-1} = KK^T$, where K is a matrix of input keys for 5000 randomly selected images. Applying the covariance matrix to the concept key k_* gives us our edit update, $(C^{-1}k_*)^T$.

2.2.2 Baseline experiments

To test how well our model editing technique generalizes to unseen scenarios, we develop partitioned test datasets for baseline experiments. We choose three baseline test datasets to experiment: random images, confusing images, and mislabelled images. For each of the three baselines, we test how editing changes the classification of our training images class "car wheel" (true positive) and how it performs on other ImageNet classes (false positive). Examples of the three baseline experiments before

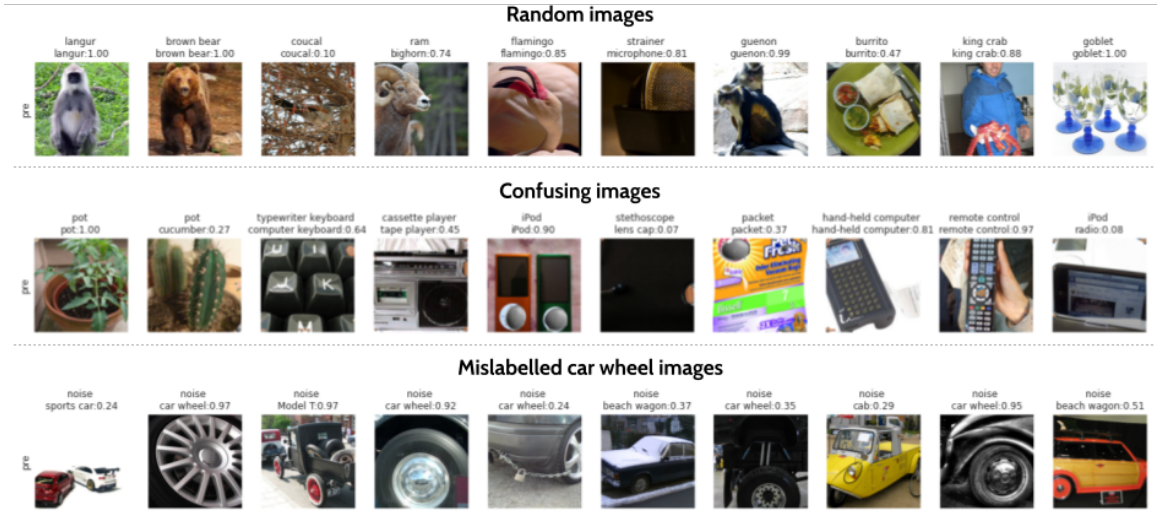

Figure 2-8: We show sample images from the three baseline test datasets created to evaluate the performance of rule editing. In the label of each image in the figure, the top line displays the ground truth class label and the bottom line displays the top predicted class along with its probability score, before editing.

editing are shown in Fig. 2-8; in the label of each image in the figure, the top line displays the ground truth class label and the bottom line displays the top predicted class along with its probability score.

Random images. We choose 5000 random images from the ImageNet validation dataset. The random images dataset allows us to evaluate the performance on classes not associated with the wheel concept, which theoretically should not be impacted by our rule editing.

Confusing images. We design confusing images that we suspect are most difficult for the network to classify correctly with this rule edit. We choose images from classes most similar to the "car wheel" class, and replace the wheel concept with the wooden wheel. We use k_* to context match for the wheel concept across all images in the ImageNet validation dataset. Of the images that context match, we select the top 1% of images by amount of wheel concept activation. For these matching images, we then overlay the pixels of the wooden wheel on the areas of the image context matched as wheel to create our confusing images, similar to the the target images of Fig. 2-7. Fig. 2-9 shows the top matching classes in our confusing

Figure 2-9: The top matching classes to the wheel concept, which are used in our confusing images test dataset, ordered from left to right by the number of images per class. Notice that our source image, the "car wheel" class has the most matching to the wheel concept, and that many of the other classes have concepts that are circular (like a wheel) and/or contain wheels.

images set, ordered from left to right by the number of images per class. Notice that many of these classes have concepts that are circular (e.g. "stethoscope", "CD player", "combination lock") or contain wheels (e.g. "minivan", "motor scooter", "minibus"), while many classes seem to have little to no relationship with wheels. The confusing images dataset allows us to evaluate the performance on classes closely related to the wheel concept, thereby testing an important aspect of the collateral effect of rule editing.

Mislabelled images. We select all the "car wheel" images from the ImageNet validation dataset, and change the ground truth label to the "noise" class. In effect, the car wheel images are all mislabelled. The mislabelled images dataset allows us to evaluate the performance on how well we can correct classifications for the "car wheel" class itself.

2.2.3 Performance

For each baseline experiment, we are interested in measuring four effects of editing: one, if "car wheel" images are still classified as "car wheel"; two, if the wooden wheels overlayed on the wheels of "car wheel" images are now recognized as "car wheel"; three, if there is collateral impact on the classes that also context match for the wheel concept; and four, if there is collateral impact on these same classes, only the images

are manipulated such that wooden wheels are overlayed on the wheels. Respectively, we refer to these as tests: normal (same class), manipulated (same class), normal (other classes), and manipulated (other classes). Note that "manipulated" refers to images where the wooden spoon is overlayed onto the wheel and "same class" refers to the "car wheel" class.

Using Fig. 2-10 and Table 2.1, we compare the performance pre- and post-editing for all four tests across all three baseline datasets. These results are shown using architecture 1 from Fig. 2-3. We highlight important takeaways from this example:

Editing using images improves classification of wheels as "car wheel." In Table 2.1, for the normal (same class) test we see that classification accuracy of car wheels increases by nearly 30%. Recall that in the normal (same class) test, we perform experiments on the original images with label "car wheel." Editing rewrites the memory bank association of wooden wheel $\rightarrow$ standard wheel, and since the remaining associations are preserved, it enables the classification as "car wheel."

Editing using images rewrites the rule as wheel $\rightarrow$ wooden wheel. As desired, Table 2.1 shows that for the manipulated (same class) test, the classification accuracy of wooden wheels as the "car wheel" class increases by nearly 50%. Recall that for the manipulated (same class) test, all of the images have a wooden texture overlayed on the wheel concept. As expected, prior to editing, the classification accuracy of just 21.95% suggests that the network has no rule, wooden wheel $\rightarrow$ standard wheel, to enable the classification of wooden wheels as "car wheel." This is likely because the network has not encountered wooden wheels before during training. The substantial increase in classification accuracy with editing suggests that editing associates wooden wheels with standard wheels, and successfully generalizes to the unseen scenario of wooden wheels.

In Fig. 2-10, we see that for the confusing images and the mislabelled car images, the post-edit classifier performs better on almost all thresholds of the ROC curve. Since performance increases with confusing images designed to share commonalities with the wheel concept, the classifier shows some specificity in that classification is

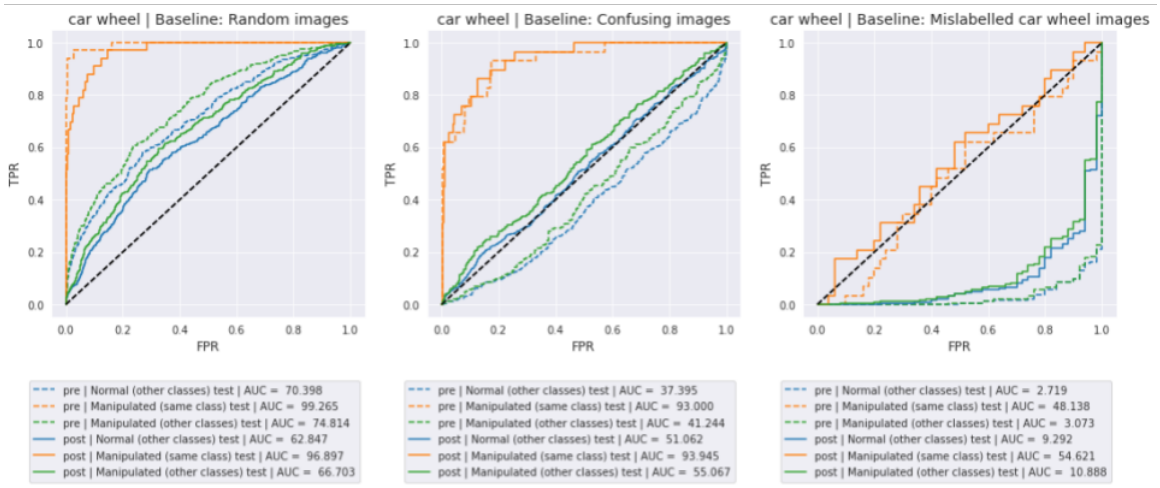

Figure 2-10: Receiver-operator curves for the three baseline test datasets, where the positive class is the "car wheel" class and the negative class is all other classes present in the test dataset.

contextual; wooden wheels map to *wheels*, instead of to similarly circular objects or other objects commonly present around wheels in images.

Editing using images still has undesirable collateral effect on other classes not associated with wheels. Recall that we designed our rewriting objective to minimize collateral effect on existing rules, and therefore preserve existing unrelated classifications. In Table 2.1, both normal (other classes) test and manipulated (other classes) test show about 20% decrease in classification accuracy for non-"car wheel" classes. This also aligns with an analysis of the post-edit ROC curves for the random images baseline, where for both tests, the post-edit classifier performs worse on almost all thresholds. Also, note that the overall accuracy of the classifier drops substantially from 69.66% to 39.08%.

2.3 Rewriting a single rule using features

In Section 2.2, we rewrite a single rule wooden wheels → conventional wheels using images. Specifically, k_* is computed from a manipulated image, where image pixels of wooden texture are overlayed on the image pixels of a wheel. In the next section, we

Test	Classification Accuracy			
	accuracy		confidence	
	pre	post	pre	post
Normal (same class) test	51.22	80.49	0.43	0.67
Normal (other classes) test	74.07	53.40	0.61	0.40
Manipulated (same class) test	21.95	70.73	0.16	0.55
Manipulated (other classes) test	56.48	33.44	0.42	0.23
All ImageNet	69.66	39.08	–	–

Table 2.1: Changes in classification accuracy and confidence across each of the four effects (tests) after editing.

present a new example that focuses on how the rewriting technique performs using features, where k_* is computed from overlaying wooden texture features onto wheel features. We also compare the performance of rewriting a rule to fine-tuning.

We introduce a new example using the MIT Places365 dataset [30], and we continue to edit the pre-trained VGG=16 model from the previous wheels example. Suppose we wish to identify whether there exist spurious correlations between the attributes of an object, such as its texture, color, and/or pattern. If so, we are interested in rewriting the classifier such that its classifications are invariant to these spurious correlations, namely object concepts of different textures, colors, or patterns. For this example, we are interested in rewriting the rule buses $\rightarrow$ stylized buses, as shown in Figure 2-11. While the structure and contextual information of the bus object are intuitively important for understanding what a bus is, in theory its paintwork is not. To the human eye, both the original bus and the stylized bus in Fig. 2-11 are undeniably buses, but such may not be the case for a network. Although all other features remain the same, by spuriously correlating the buses' paintwork with the concept, it may cause targeted misclassifications.

To identify this spurious correlation, we produce manipulated bus images. We use a pre-trained COCO segmentation model [21] to segment ImageNet images for the bus concept, and style transfer [9][18] a desired pattern (in this example, gravel) onto the segmented bus. We refer to these manipulated bus images as stylized images.

Figure 2-11: Does the network spuriously correlate the buses' paintwork with the concept of the bus? We present stylized versions of the bus concept, where the paintwork of the bus is replaced with a gravel pattern.

With this technique, we can exclusively change the object we segment for; notice that the wheels, lights, and all other parts of the image remain the same.[4]Similar to the wooden wheel images from the previous example, the network has never seen these stylized bus images before.

2.3.1 Using features to compute k_* and v_*

Recall that k_* and v_* are vector representations of the concepts we wish to edit from and to, respectively. In Section 2.2.1, we compute k_* from the photo-shopped image of the wooden spoon overlayed onto the wheel. We pass the photo-shopped image through the model, and use the input representation of the layer to compute k_*. In effect, we are copying the *pixels* of the wooden texture and pasting it onto the pixels of the wheel, then context matching for the wooden wheel feature representation. In this section, we compute k_* by copying the *features* of the stylized pattern and pasting it onto the features of the bus.

In theory, with this method the network should be more receptive to editing only

[4]Certainly, it is important to note the precision with which we exclusively stylize the desired object is limited by the performance of the pre-trained COCO segmentation model.

Test Dataset	Number of images	Misclassification error rate w.r.t. Ground truth label	Misclassification error rate w.r.t. Pre-edit model pred
Target class	72	97.14%	95.83%
Most commonly misclassified classes	93	62.96%	68.82%

Table 2.2: The most commonly misclassified classes (not including the target class, "bus station/indoor") after applying the stylized bus to all ImageNet images. We record the significant drop in accuracy. We test how well our editing technique can rectify these mistakes using these two test datasets.

the desired concept (thereby reducing collateral impact) because we are exclusively transplanting the relevant features of the style.[5] In contrast, in using images to context match, the network has never seen images of stylized buses before and very likely does not have a robust feature representation for these stylized buses. Moreover, there is no need for us to create new photo-shopped images, such as the target images in Fig. 2-7, to compute k_*.

2.3.2 Investigating a spurious correlation

Since we are primarily interested in identifying whether the spurious correlation causes the classifier to misclassify the stylized bus across all classes, we generate a new test dataset that uses context matching and concept segmentation to replace buses with the stylized version. Fig. 2-12 shows the impact of replacing the bus with the stylized bus, by the rate of errors per class as well as the average change in probability of the top prediction per class. As expected, our target class – "bus station/indoor" – is most impacted, according to both the number of misclassifications and the severity of misclassifications. Notice, however, that classes seemingly unrelated to the bus concept (e.g. "laundromat", "phone booth") are also impacted by the stylized images.

The reduction in classification accuracy across classes suggests that there is indeed a spurious correlation between the paintwork of the bus and the bus concept. In this instance, we have identified the spurious correlation because of a distribution shift in

[5]In this example, the style we are transferring is gravel, which the network has seen before in the context of other classes. Other styles with which we tested, such as floral patterns and fuzzy textures, have also been seen by the network. In theory, the layer may have an existing understanding of the stylized concepts in feature space.

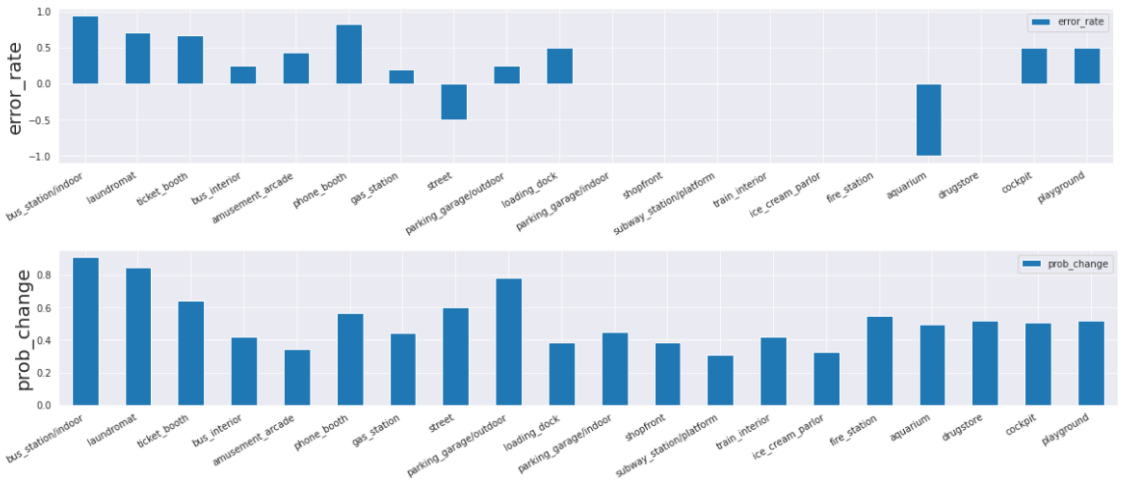

Figure 2-12: After replacing all buses across all Places365 classes with the stylized buses, we measure the change in error rate and the average change in top probability score for each class.

the training and test datasets. Additionally, to test how well we can mitigate this spurious correlation, we filter our test data with which we evaluate our technique, to these most commonly misclassified classes, explained in Table 2.2. In Fig. 2-13, we show examples of instances in which the classifier classifies the original image of the bus correctly, but misclassifies the manipulated image with the stylized bus.

2.3.3 Fine-tuning

One of the most common retraining techniques, fine-tuning, modifies the parameters of a pre-trained model to perform in another task [20]. In our example, we can fine-tune the layer weights to learn to understand the stylized bus concept to correct the misclassifications of the stylized bus. Since both our rewriting technique and fine-tuning attempt to preserve existing representations while simultaneously incorporating new representations, we compare the performance of our rewriting technique with fine-tuning.

To fine-tune, similar to our rewriting technique, we freeze all weights except for those of the desired layer to edit, l. Freezing part of the network also prevents overfitting. To find the fine-tuned weights of l, W^{ft}, we define our objective function

Figure 2-13: Examples of stylized bus images in which the classifier classifies the original image of the bus correctly, but misclassifies the manipulated image with the stylized bus. For each manipulated image, the top row shows the ground truth label, the middle row shows the pre-trained model's prediction for the original image, and the bottom row shows the pre-trained model's prediction for the manipulated image.

for fine-tuning as:

$$W^{ft} = \arg\min_{W} \sum_{i} CE\left(h(W * X_i); Y_i\right) \tag{2.22}$$

where W denotes the weights of the desired layer l, $CE(\cdot)$ indicates cross entropy loss, X_i denotes the features computed by the previous $l-1$ layers for image i, $h(\cdot)$ describes all the layers after l, and Y_i is the ground truth label for image i. All images i are manipulated images of the stylized bus. We use stochastic gradient descent to perform the optimization.

2.3.4 Performance

Our experiments are designed to evaluate whether our rewriting technique is an effective method to edit the pre-trained network to mitigate mistakes in classifications made by spurious correlations. We compare the performance of three ideas: rewriting

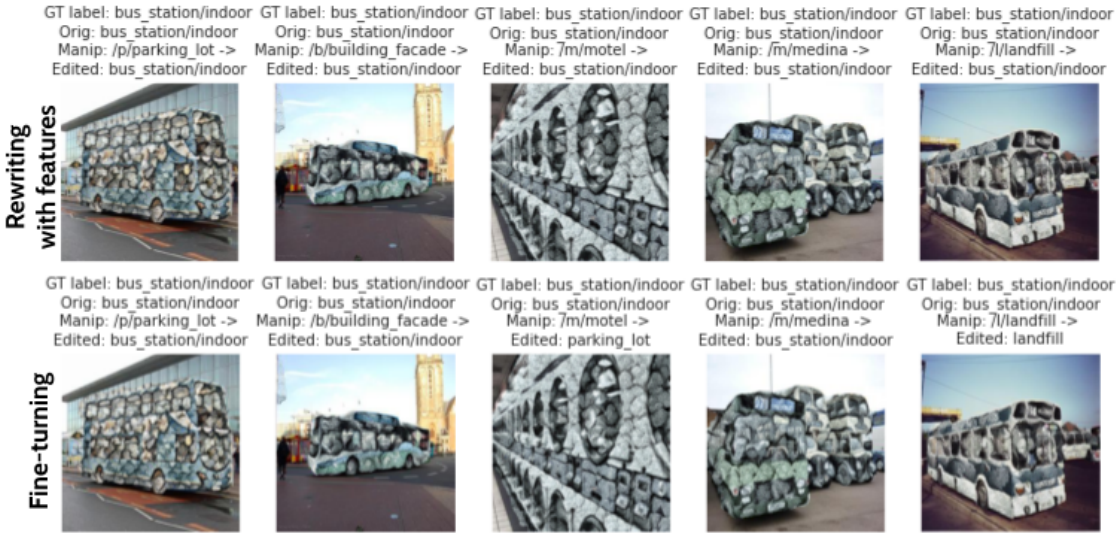

Figure 2-14: For both rewriting with features technique and fine-tuning, we show the same five testing dataset images. For each image, the top row is the ground truth label, the second row is the original model's prediction for the unedited bus image, the third row is the original model's prediction for the stylized bus image (displayed), and the fourth row is the edited model's prediction for the stylized bus image.

using features, rewriting using images, and fine-tuning. We use the filtered test data detailed in Section 2.3.2 for all evaluation. The results of the accuracy changes for the target class, "bus indoor", are shown in Table 2.3 and averaged for all other classes are shown in Table 2.4. We highlight the following conclusions:

Rewriting corrects mistakes in classification more comprehensively than fine-tuning does. Both rewriting techniques correct significantly more errors than the fine-tuning technique for the target class, approximately $\geq 60\%$ more corrections, and for the most misclassified classes, $\geq 20\%$ more corrections. Fig. 2-14 shows examples of corrections for rewriting with features and for fine-tuning. Notice that for the third and fifth image, while rewriting with features correctly rewrites the classification, fine-tuning does not. As desired with reducing the effects of spurious correlations present in the original model, rewriting also corrects more errors with respect to the ground truth label than with respect to the original model prediction.

	Target class				Overall ImageNet accuracy
	Accuracy w.r.t Ground truth label	Errors corrected w.r.t. Ground truth label	Accuracy w.r.t Pre-edit model pred	Errors corrected w.r.t. Pre-edit model pred	
pre	2.86%	–	4.17%	–	54.03%
rewriting using features	80.00%	79.41%	80.56%	79.71%	45.60%
rewriting using images	90.00%	89.71%	88.89%	88.41%	43.44%
fine-tuning	22.86%	20.59%	23.61%	20.29%	53.96%

Table 2.3: Changes in accuracy for the target class, "bus station, indoor" with respect to the ground truth label and the original prediction by the pre-trained model. We compare the performance across rewriting using features, rewriting using images, and fine-tuning.

Compared to rewriting with images, rewriting with features reduces collateral impact on other classes, but reduces performance on target class. As hypothesized, compared to rewriting with images, rewriting with features reduces collateral impact on other classes. In Table 2.4, rewriting using features corrects ∼6% more errors with respect to the ground truth label, and ∼2% more errors with respect to the original model prediction for other classes. A likely cause is that rewriting with features better context matches for the stylized concept, since it uses representations the network already has. Rewriting with features also more exclusively (although, still with notable noise) transplants the relevant features of the style variant, reducing potential noise from other parts of the image.

However, we see a marked reduction in classification accuracy for the the target class. In Table 2.3, rewriting using features corrects ∼10% fewer errors with respect to the ground truth label, and ∼10% fewer errors with respect to the original model prediction for other classes. This may be caused in part because in pasting the variant features onto the original concept, we are replacing some of the critical features unique to the original concept, which the variant features do not capture. These critical features may be most useful in correct classification.

Rewriting has considerable unintended impact on other existing rules. Certainly the most considerable limitation of rewriting is that the the cost is a marked drop of ∼10% in overall accuracy of all ImageNet classes. It suggests that our rewriting technique causes considerable collateral damage on existing rules.

	Other classes				Overall ImageNet accuracy
	Accuracy w.r.t Ground truth label	Errors corrected w.r.t. Ground truth label	Accuracy w.r.t Pre-edit model pred	Errors corrected w.r.t. Pre-edit model pred	
pre	37.04%	–	31.18%	–	54.03%
rewriting using features	59.26%	35.29%	50.54%	28.12%	45.60%
rewriting using images	55.56%	29.41%	49.46%	26.56%	43.44%
fine-tuning	42.59%	8.82%	37.63%	9.38%	53.96%

Table 2.4: Changes in accuracy for non-target classes, with the most misclassifications from replacing the bus with the stylized version. Classification accuracies are shown with respect to the ground truth label and the original prediction by the pre-trained model. We compare the performance across rewriting using features, rewriting using images, and fine-tuning.

2.4 Discussion

With minimal, but powerful intervention to the network, we are able to use our rewriting technique to (1) generalize to unseen scenarios, (2) identify spurious correlations by enforcing a distribution shift between training and testing datasets, and (3) successfully rewrite an existing rule to correct for misclassifications caused by spurious correlations. Although we may not see wooden wheels or gravel-stylized buses in reality, these examples demonstrate the endless ability to use human intervention to learn about and correct undesirable observations in commonly used pre-trained networks. Most notably, our technique forgoes the need for more than one new training images or the need to re-train most of the network.

Editing multiple rules at once. Although this work focus on editing a single rule, it is a simple extension of our current problem formulation to edit multiple rules at once. To alter i rules at once, we define our optimization as:

$$\lambda_D = \arg\min_{\lambda} ||V_D - l(K_D; W^0 + \lambda(C^{-1}K_D)^T)|| \tag{2.23}$$

where K_D is a matrix of keys and each row contains the update direction $C^{-1}k_i$, λ_D is a matrix of Lagrangian multipliers, and V_* is a matrix of values. Once we find λ_D, we can update our weights as such:

$$W_D = W_0 + \lambda_D(C^{-1}K_D)^T \tag{2.24}$$

Using images versus features to synthesize context match. In our example of using features to produce the key that context matches for stylized buses, we rely on the observation that the network has seen the gravel style before and therefore has an internal feature representation for gravel. However, it may not always be the case that the network has an existing representation for an unseen variant. In this case, it may be better to use images, as we did in the wheel example, to construct a useful representation.

A more comprehensive comparison across different concepts, datasets, and models. Certainly more work needs to be done to thoroughly understand the effectiveness of the rewriting technique, specifically across various visual concepts, datasets, and models. Preliminary work suggests that our rewriting technique does successfully edit pre-trained ResNet models [16].

Limitations to our rewriting technique. As we observed from our performance evaluation of our rewriting technique, we still see notable collateral damage on existing rules. In practice, we assume that we have an error-free memory, where the keys form an orthogonal set. In reality, this is not true and ultimately contribute to the collateral effect.

Chapter 3

Editing a Classifier in the Inverse Direction

Suppose that at layer l with weights W, our input to the layer $A^{[l-1]}$ has $c-$input channels and our output of the layer $A^{[l]}$ has $f-$output channels. Suppose we treat this as a linear operation, then we can write the layer as:

$$A^{[l]} = WA^{[l-1]}$$

In the forward direction, our key $\in A^{[l-1]}$ has $c-$channels and our value $\in A^{[l]}$ has $f-$channels. As we progress deeper through our convolutional layers in VGG16, we increase the number of channels to capture increasing complexity. So, we can state that $f >= c$. Recall that in Section 2.1.1, we motivate considering the rewriting problem in the inverse direction. Specifically, since our classifier reduces dimensionality, perhaps a dense matrix that also reduces dimensions by mapping $f-$input channels to $c-$output channels would better align in the classifier setting. Notice that in this new mapping of $f-$input channels to $c-$output channels, we are switching what we consider the input and the output. Therefore, in the inverse direction, our key has $f-$channels and our value has $c-$channels. Since our key $\in A^{[l]}$ and our value $\in$

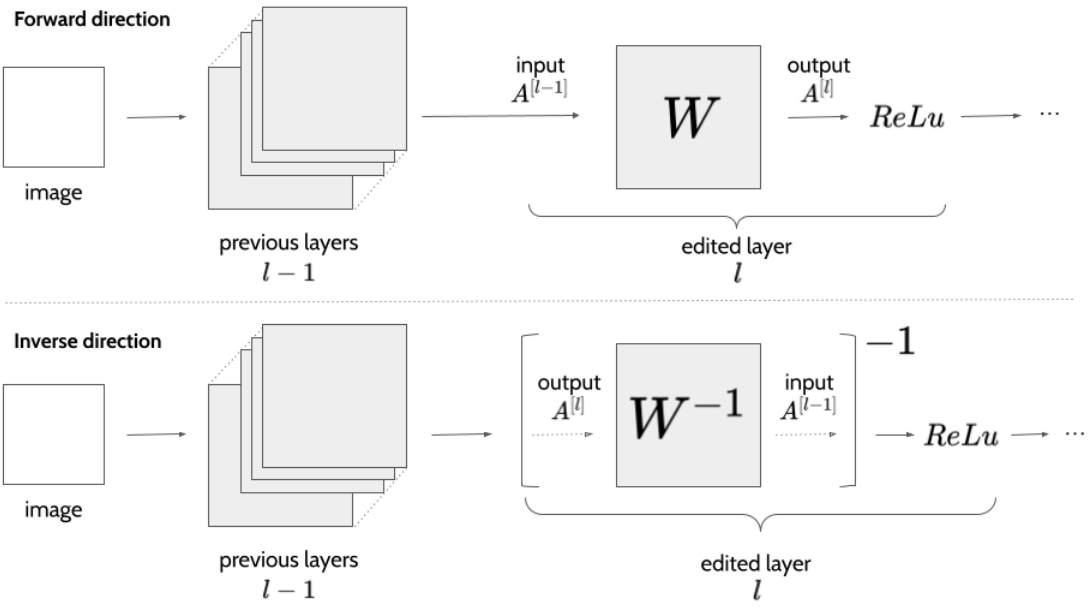

Figure 3-1: In rewriting, we focus on editing the weights of layer l only. There are two ways to think about how we are editing: the forward direction and the inverse direction. We formulate editing in the inverse direction as it might align better with the classifier setting.

$A^{[l-1]}$, we can express this switch as:

$$A^{[l-1]} = W^{-1}A^{[l]}$$

where, assuming W is invertible, W^{-1} is the inverse of W. To perform the edit, we minimize our loss function on W^{-1}. We then invert the updated weights and insert it back into the network so that the input and output channel sizes are now the same as the unedited network. We present both the forward and inverse directions diagrammatically in Fig. 3-1. Notice that the key difference is that instead of operating on W as we did in the forward direction, in the inverse direction we operate on the inverse, W^{-1}.

This chapter is an exploration of another interpretation of the rewriting technique, the inverse direction. We experiment with various formulations for editing in the inverse direction, and hope that future directions of this thesis continue to expand on

the inverse direction.

3.1 Operating on W^T instead of W^{-1}

Our formulation in Eq. 3.1, where we operate on W^{-1} is problematic for two reasons. First, notice that W is very likely not a square matrix, in which case there exists no regular two-sided inverse. Second, even if W were invertible, it may be computationally intensive to compute (especially if we were to perform multiple rounds of edits).

In the inverse direction, what we are truly interested in with regards to updating our weights is the term that propels backpropagation in the forward direction. We know the following to be true about the forward direction:

$$A^{[l]} = WA^{[l-1]} \tag{3.1}$$

Suppose we wish to edit the rule $A_*^{[l-1]} \rightarrow A_*^{[l]}$, where $A_*^{[l-1]}$ is our key and $A_*^{[l]}$ is our value. Suppose our loss function is squared error:

$$L = (A^{[l]} - A_*^{[l]})^2 \tag{3.2}$$

Since we are interested in what fuels backpropagation in the forward direction, we solve for $\frac{\partial L}{\partial A^{[l-1]}}$:

$$\frac{\partial L}{\partial A^{[l-1]}} = \frac{\partial L}{\partial A^{[l]}} \cdot \frac{\partial A^{[l]}}{\partial A^{[l-1]}} \tag{3.3}$$

$$= 2(A^{[l]} - A_*^{[l]}) \cdot W^T \tag{3.4}$$

$$= 2W^T A^{[l]} - 2W^T A_*^{[l]} \tag{3.5}$$

In our optimization, we can assume that the initial guess $A^{[l]}$ is random, so in expectation, $2W^T A^{[l]} = 0$. Thus, we can say that W^T directs $A_*^{[l-1]}$ towards $A_*^{[l]}$. Therefore, to perform the edit where we update the weights, we operate on W^T. Computationally, this become much more feasible.

3.1.1 Method

Suppose we want to edit the rule wooden wheels → conventional wheels, where wooden wheels is our key concept and conventional wheels is our value concept, as we did in Section 2.2. Intuitively, we want W^T to guide the conventional wheel to behave like a wooden wheel.

To edit the desired layer l, we use $A^{[l-1]}$ to denote the features computed by the first $l-1$ layers of the network, and $A^{[l]} = l(A^{[l-1]}; W_0^{[l]})$ to denote the computation of layer l, which has pretrained weights, $W_0^{[l]}$. To edit in the inverse direction, we edit the transpose of the weights, W^T. In other words, we now consider the computation of layer l as $v = l(k; W_0^T)$. We wish to assign a single key k_* to a new value v_*. Our objective function is:

$$W_1 = \left[\arg\min_W \mathcal{L}_s(W) + \lambda \mathcal{L}_c(W)\right]^T \tag{3.6}$$

$$\mathcal{L}_s(W) := \mathbb{E}_k \left[||l(k; W_0^T) - l(k; W^T)||^2\right] \tag{3.7}$$

$$\mathcal{L}_c(W) := ||v_* - l(k_*; W^T)||^2 \tag{3.8}$$

where $||\cdot||^2$ denotes L2-loss.

In Algorithm 1, we present how to edit in the inverse direction using the optimization presented in Eq. 3.6:

Algorithm 1 Procedure to edit in inverse direction

Input: W_0 (pre-trained weights), $A_k^{[l-1]}$ (input representation of key in rule edit, $A_v^{[l-1]}$ (input representation of value in rule edit)
Output: W_1 (edited weights)

procedure
  $A_v^{[l]} \leftarrow W_0 A_v^{[l-1]}$
  $k_* \leftarrow A_v^{[l]}$
  $v_* \leftarrow A_k^{[l-1]}$
  function $\mathcal{L}(v_*, l(k_*; W^T))$ ▷ optimization Eq. 3.6
    return W^T
  end function
  $W_1 = (W^T)^T$ ▷ transpose of optimization Eq. 3.6
  return W_1
end procedure

There are two important things to notice here: one, unlike in the forward direction, k_* is based on the representation of the conventional wheel and v_* is based on the representation of the wooden wheel; and two, the optimization occurs on W^T, so for the dimensions to align in the original network, we transpose the results of the optimization to be our newly edited weights.

Image inversion to visualize features

To evaluate whether rewriting works, we first pass an image of a wheel through the edited network. We then perform gradient descent optimization to compute the image, initialized at random, that reconstructs the target output of the layer for the wheel image. Suppose the wheel image produces a target output $T \in A^{[l]}$ and F represents all the layers up to and including l. We can reconstruct an image x using the loss function:

$$x_{\text{recon}} = \arg\min_x \mathcal{L}_{\text{image}}(x) \tag{3.9}$$

$$\mathcal{L}_{\text{image}}(x) = ||F(x) - T||^2 \tag{3.10}$$

where $||\cdot||$ is L2-loss. We can then visualize the reconstructed image. These image reconstructions are synonymous to image inversions in robust models, introduced in used [7], for their representation inversion property.

Before editing, since the rule maps wheels $\rightarrow$ wheels, the reconstructed car image should look the same; the wheels on the car still behave like wheels. In theory, with editing, since the rule maps wooden wheels $\rightarrow$ wheels, the reconstruction of the same car image should have wooden wheels replacing the conventional wheels. This is because the rule edit enforces that wheels should now behave like wooden wheels.

3.1.2 Results

We find that operating on the transpose does not sufficiently rewrite the desired rule in the inverse direction. In Fig. 3-2, we mask the location of the wheels to produce

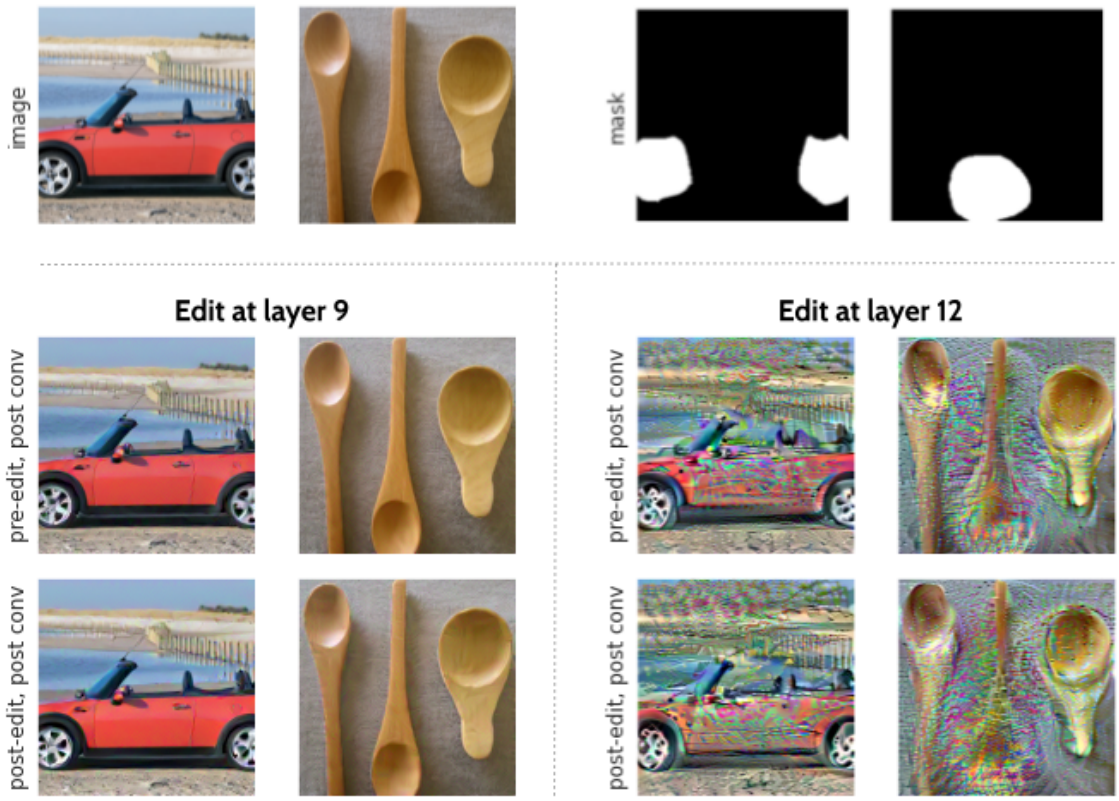

Figure 3-2: We enforce the edit, wheels → wooden wheels in the inverse direction, in one experiment at layer 9 and in another at layer 12, and visualize the image reconstructions.

the wheel key and we mask the location of the spoon to produce the wooden texture concept. We then enforce the edit, wheels $\rightarrow$ wooden wheels in the inverse direction, in one experiment at layer 9 and in another at layer 12[1], and visualize the image reconstructions. We conclude that since the image reconstructions remain the same, such that wheels still behave like wheels, editing W^T does not enforce the rule edit.

Since operating on W^T is insufficient, we return to the drawing board to ideate on what else may be a sufficient approximation of W^{-1} to edit on.

3.2 Operating on $W^\dagger$ instead of W^{-1}

To approximate the inverse, we use the most widely known generalization of the inverse matrix, the Moore-Penrose inverse, also known as the psuedoinverse. Instead of operating on W^{-1}, we operate on the psuedoinverse, $W^\dagger$.

3.2.1 Method

To edit the desired layer l, we use $A^{[l-1]}$ to denote the features computed by the first $l-1$ layers of the network, and $A^{[l]} = l(A^{[l-1]}; W_0^{[l]})$ to denote the computation of layer l, which has pretrained weights, $W_0^{[l]}$. To edit in the inverse direction, we edit the psuedoinverse of the weights, $W^\dagger$. In other words, we now consider the computation of layer l as $v = l(k; W_0^\dagger)$. We wish to assign a single key k_* to a new value v_*. Our objective function is:

$$W_1 = \left[\arg\min_W \mathcal{L}_s(W) + \lambda \mathcal{L}_c(W)\right]^\dagger \tag{3.11}$$

$$\mathcal{L}_s(W) := \mathbb{E}_k\left[||l(k; W_0^\dagger) - l(k; W^\dagger)||^2\right] \tag{3.12}$$

$$\mathcal{L}_c(W) := ||v_* - l(k_*; W^\dagger)||^2 \tag{3.13}$$

[1]Layer 9 is the convolutional layer prior to the second to last max pooling layer, and layer 12 is the last convolution layer of the network. All experiments in this chapter operate using architecture 4 from Fig. 2-3.

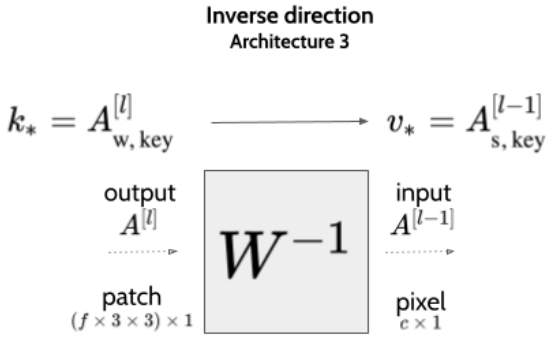

Figure 3-3: For architecture 3 in Fig. 2-3, we edit from a patch of size 3×3 to a single pixel. If editing wheels to behave look wooden wheels (or, equivalently spoons), then k_* is the same size as our output of the layer, $A^{[l]}$, and v_* is the same size as our input to the layer, $A^{[l-1]}$.

where $||\cdot||^2$ denotes L2-loss. We use a modified version of Algorithm 1, where instead of performing the optimization using Eq. 3.6, we use Eq. 3.11.

Dot product similarity to visualize concept similarity

Recall that our goal is for the wheels to behave like wooden wheels (or, equivalently as spoons since the wooden texture is derived from a spoon). Fig. 3-3 show us the edit $k_* \rightarrow v_*$ visually. To evaluate whether rewriting works, we want to visualize if and how similar the wheels are to spoons, pre- and post-editing. We ask: what is the best input layer reconstruction, $A^{[l-1]}_{\text{w, recon}}$, of the output layer representation, $A^{[l]}_w$? Visualizing $A^{[l-1]}_{\text{w, recon}}$ allows us to see the effects of editing in the inverse direction.

To do so, we first pass an image of a wheel through the edited network. The wheel image produces a target output at layer l of $A^{[l]}_w$ and input at layer l of $A^{[l-1]}_w$. We can reconstruct $A^{[l-1]}_{\text{w, recon}}$, which is initialized at random, from $A^{[l]}_w$ using the loss function:

$$A^{[l-1]}_{\text{w, recon}} = \arg\min_{A^{[l-1]}} \mathcal{L}_{\text{features}}(A^{[l-1]}) \tag{3.14}$$

$$\mathcal{L}_{\text{features}}(A^{[l-1]}) = ||l(A^{[l-1]}) - A^{[l]}_w||^2 \tag{3.15}$$

where $||\cdot||$ is L2-loss. We can overlay the reconstructed wheel representation as a heatmap onto the original wheel image, to visualize the salient features encoded in

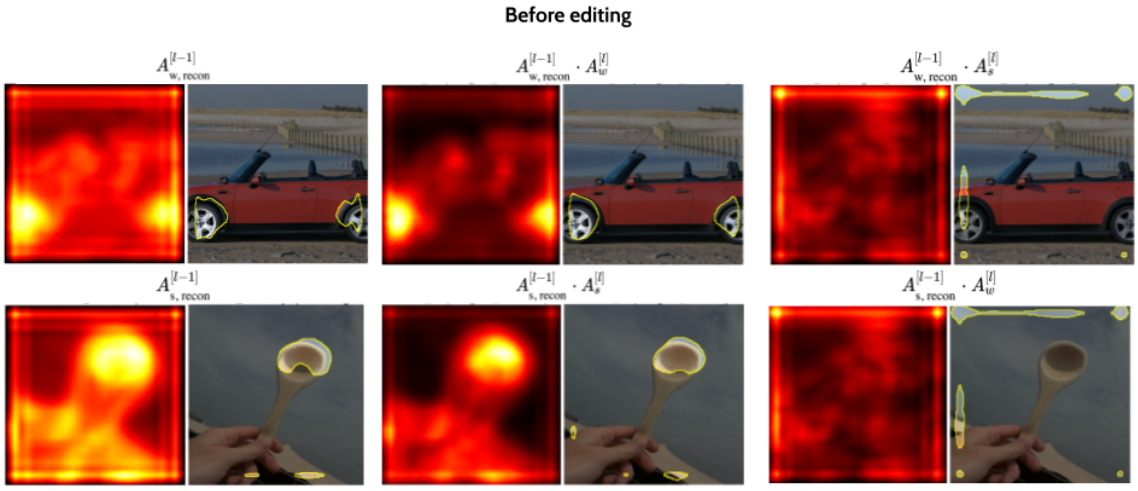

Figure 3-4: Before any editing is performed, we show the heatmap of the reconstructed representations and the pairwise dot product similarity between the reconstructed representation and the target representation. We also overlay the heatmaps onto the original image used to create the reconstruction.

the representation.

Theoretically, if editing succeeds, then reconstructed wheel feature representation should be similar to a spoon feature representation. To find this output representation of a spoon, we pass an image of a spoon through the edited network, which produces an output at layer l, $A^{[l]}_s$. So, if editing succeeds, then $A^{[l-1]}_{\text{w, recon}}$ should be similar to $A^{[l]}_s$. We quantify similarity between two representations by calculating the dot product similarity between corresponding pairs of points between the representations. We can also overlay the dot product similarity onto the original wheel and spoon images to visualize the location of similarity (or lack thereof).

We can also perform the same reconstruction using the spoon image to produce $A^{[l-1]}_{\text{s, recon}}$. Theoretically, if editing succeeds, then the reconstructed spoon should still remain similar to a spoon, so $A^{[l-1]}_{\text{s, recon}}$ should be similar to $A^{[l]}_s$. Fig 3-4 shows us that before editing, as expected, that $A^{[l-1]}_{\text{w, recon}} \approx A^{[l]}_w$ and $A^{[l-1]}_{\text{s, recon}} \approx A^{[l]}_s$, while there is little to no similarity between $A^{[l-1]}_{\text{w, recon}}$ and $A^{[l]}_s$. Our goal in this example is that after editing, $A^{[l-1]}_{\text{w, recon}} \approx A^{[l]}_s$.

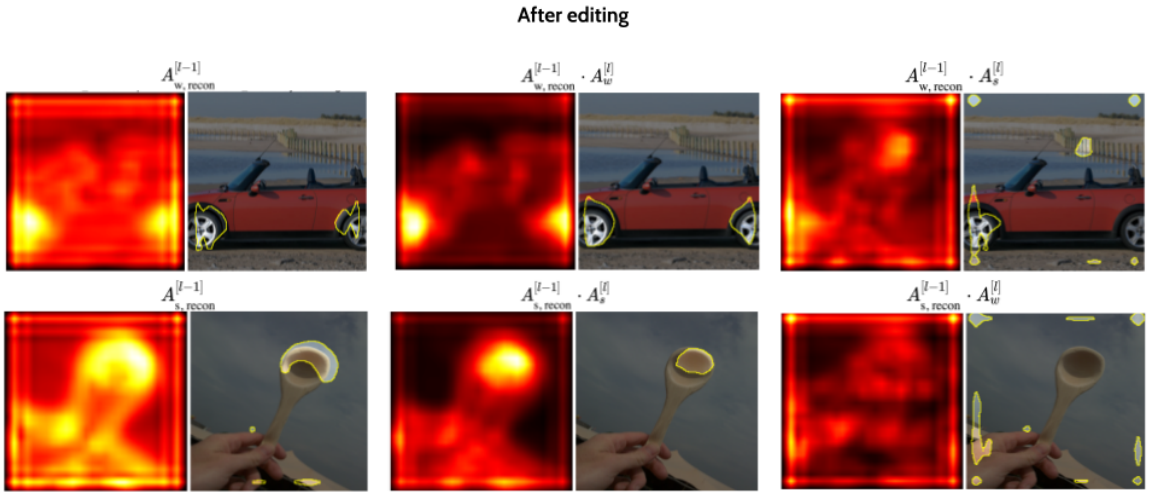

Figure 3-5: After editing, same heatmaps as shown in Fig. 3-4.

3.2.2 Results

We wish to perform the edit, $k_* = A^{[l]}_{\text{w, key}} \rightarrow v_* = A^{[l-1]}_{\text{s, key}}$ where *key* denotes a vector.[2] We perform the edit outlined in Eq. 3.12 at layer 9, with the desired goal that wheels behave like wooden wheels. Fig. 3-5 shows the same heatmaps as Fig. 3-4, but after we perform our edit. Notice that as desired with editing, $A^{[l-1]}_{\text{w, recon}}$ is closer in similarity to $A^{[l]}_s$ than before, and that most of the similarity with the spoon concept is located around the wheel pixels. However, there is still some undesired similarity with the spoon around the spoon pixels, suggesting that the edit is not localized to the wheel pixels. In addition, Fig. 3-6 shows that we are unsuccessful in producing a substantial enough rule rewrite, such that the wheels in the image reconstruction look visually like wooden wheels.

[2]In other words, the feature maps have spatial information $q \times r$, while the keys do not. For example, if $A^{[l-1]}$ has dimensions $c \times q \times r$ and $A^{[l]}$ has dimensions $f \times q \times r$, then for a 3×3 convolution: $A^{[l]}_{\text{w, key}}$ has dimensions $(f \times 3 \times 3) \times 1$. And, $A^{[l-1]}_{\text{w, key}}$ has dimensions $c \times 1$. Again, note that this corresponds with architecture 3 of Figure 2-3, where we map from $f \times 3 \times 3-$channels to $c-$channels.

Figure 3-6: After editing, image inversions of the same wheel and spoon images.

So, in the process of editing with keys, somewhere in the pipeline of visualizing the reconstructions of our keys, feature maps, and images, the wheel is not behaving sufficientiyl like a sufficiently. The following must be true at each stage in our visualization pipeline for editing to be considered successful:

- **Keys.** The reconstruction using a single given pixel of the wheel key is approximately equal to a patch of pixels of the spoon key, $A^{[l-1]}_{\text{w, recon, key[pixel]}} \approx A^{[l]}_{\text{s, key}}[\text{pixel} : \text{pixel} + 9]$.

- **Feature maps.** At the pixel of a wheel on the feature map, the reconstruction of the wheel representation is approximately equal to a patch of pixels of the spoon representation, $A^{[l-1]}_{\text{w, recon, pixel}} \approx A^{[l]}_{\text{s}}[\text{pixel} : \text{pixel} + 9]$.

- **Images.** At the pixels of wheels on the image, the reconstruction of the wheel image is approximately equal to the spoon image, $x_{\text{w, recon}} \approx x_{\text{s}}$.

For each of these stages, we record the distance between the reconstructed object and each target object (i.e. wheel or spoon) at each iteration of reconstruction loss. Theoretically, both before and after editing, since the reconstructed object is initialized at random, it is fairly similar in distance to both the wheel and spoon. Before editing, as we progress through optimization we should see that the the reconstructed object looks more and more like a wheel. After editing, the reconstructed object should look more and more like a spoon. For the keys and feature maps, we chose cosine similarity as our distance metric to normalize for vector magnitude. The higher the cosine similarity, the more similar the inputs are. For the images, we chose LPIPS

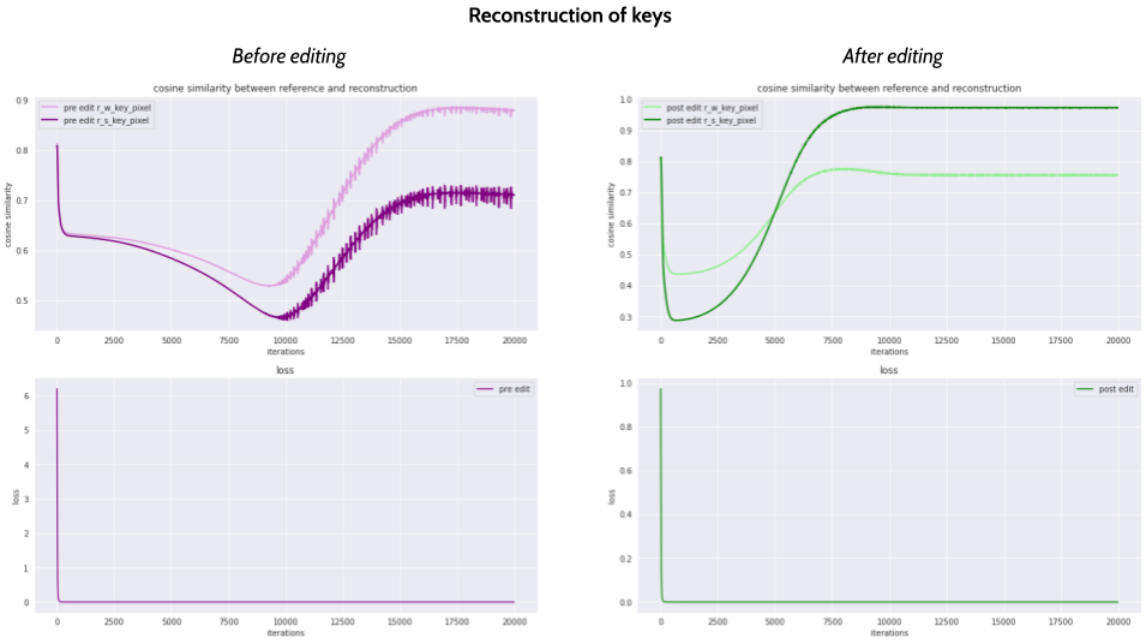

Figure 3-7: During reconstruction of keys, plot of loss over reconstruction iterations and plot of cosine similarity distance between reconstructed key and target wheel/spoon key. Note that cosine similarity for all curves does converge.

distance [28] as our distance metric, since it measures perceptual similarity between two images. The lower the LPIPS distance, the more similar the inputs are.

Keys. Fig. 3-7 shows that before editing, the reconstruction looks closer to the wheel (light purple curve) than it does to a spoon (dark purple curve). After editing, the reconstruction looks closer to a spoon (dark green curve) than to the wheel (light green curve). So, at the key stage, we are successfully editing the wheel to behave like a spoon.

Feature maps. Fig. 3-8 shows that before editing, the reconstructed feature map is undoubtedly similar to the wheel. After editing, the reconstructed feature map is still much more similar to the wheel, although notably less so and more closer to the spoon.

Images. Fig. 3-9 quantifies what we qualitatively see in Fig. 3-6: after editing, the reconstructed images remain the same. Editing does not produce the desired effect of wheels behaving and looking like spoons.

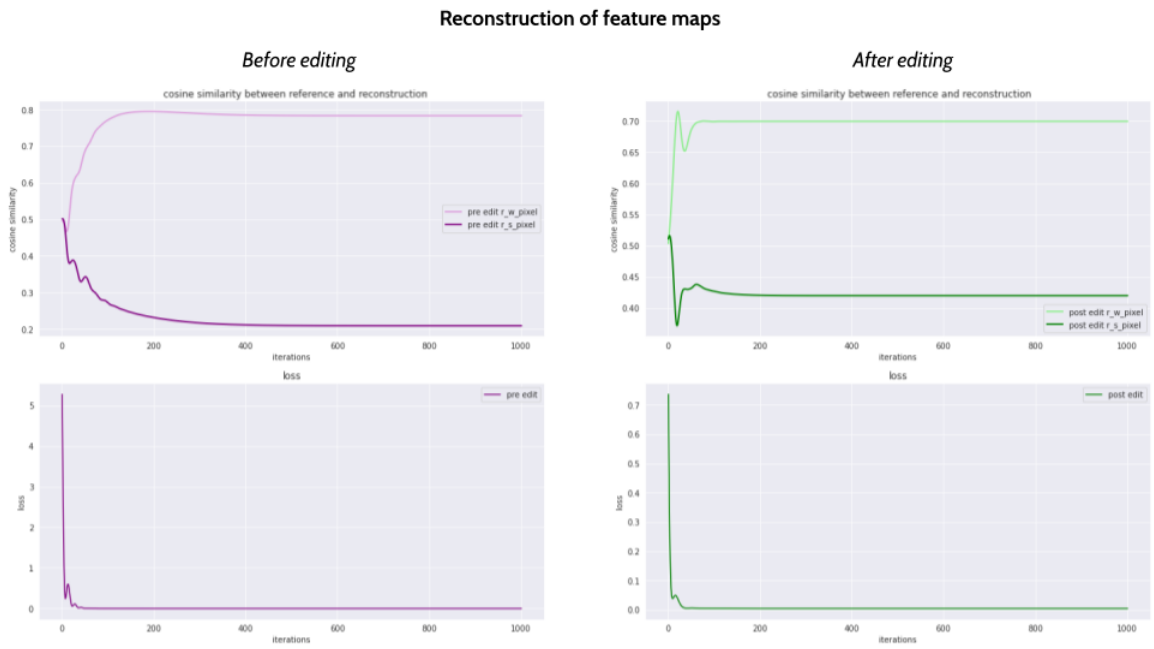

Figure 3-8: During reconstruction of feature maps, plot of loss over reconstruction iterations and plot of cosine similarity distance between reconstructed feature map and target wheel/spoon feature map. The reconstruction is constructed from a single pixel of the wheel on the feature map. Note that cosine similarity for all curves does converge.

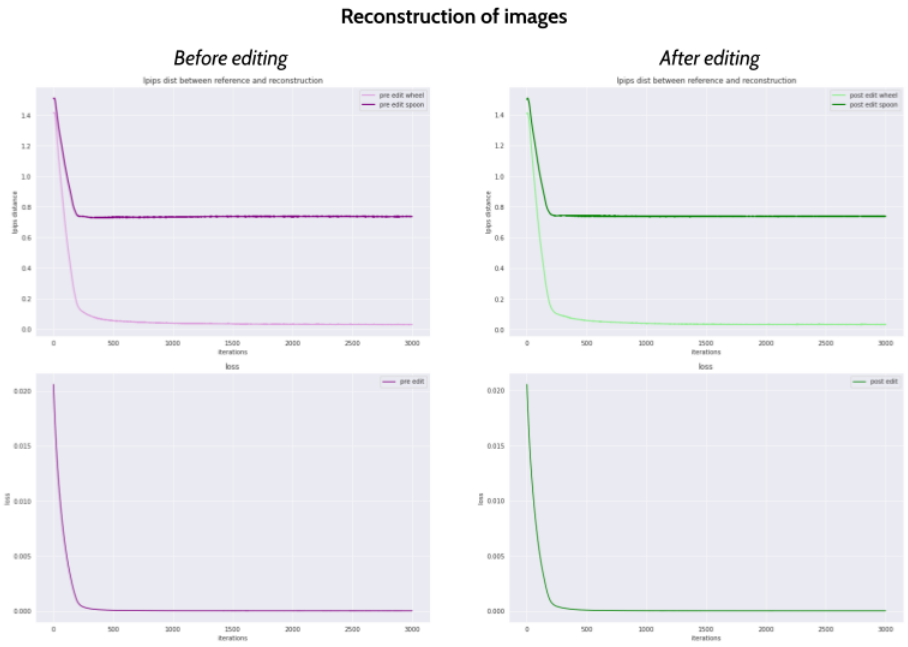

Figure 3-9: During reconstruction of images, plot of loss over reconstruction iterations and plot of LPIPS distance between reconstructed image and target wheel/spoon images. Note that LPIPS distance for all curves does converge.

From our reconstructions, we learn that editing works for a single pixel of the key, but not for a field of pixels of the featuremap. So, we hypothesize that there is some interaction between adjacent pixels, which dominates in our edit. Recall from Section 3.3.2 that we generalized our rule edit to a nonlinear, convolutional layer by editing the linear operation W. Perhaps this solution ignores the convolutional interactions that are fundamental to successful editing.

3.3 Treating W as a convolutional matrix

Since l is a convolutional layer, represented in Fig. 3-10, with a fixed stride of 1 and a padding of 1, we write the forward pass of l, which convolves the input to the layer

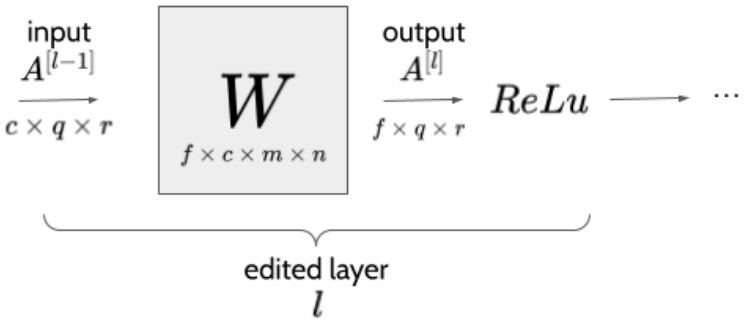

Figure 3-10: Representation of edit to layer l. W denotes the weights of the layer, $A^{[l]}$ denotes the output features computed by layer l, and $A^{[l-1]}$ denotes the input features computed by the previous $l-1$ layers. Note that the dimensions of the kernel are $m \times n$, the channel number is c, the filter number is f, and the dimensions of the input and output feature maps are $q \times r$.

with the weights using the convolutional operator $\circledast$:

$$A^{[l]}_{f,q,r} = (W \circledast A^{[l-1]})_{f,q,r} \tag{3.16}$$

$$= \sum_{c} \sum_{m,n=-1}^{1} A^{[l-1]}_{c,q-m,r-n} W_{f,c,m,n} \tag{3.17}$$

where W denotes the weights of the layer, $A^{[l]}$ denotes the output features computed by layer l, and $A^{[l-1]}$ denotes the input features computed by the previous $l-1$ layers. Note that the dimensions of the kernel are $m \times n$, the channel number is c, the filter number is f, and the dimensions of the input and output feature maps are $q \times r$. We also ignore the non-linear activation function and bias term.

3.3.1 Least squares solution

In Section 2.1.4, we used the following objective function with linear operations to find our pre-trained weights W^0:

$$W^0 = \arg\min_{W} ||V - WA^{[l-1]}||^2 \tag{3.18}$$

We can rewrite our objective function with convolutional operators:

$$W^0 = \arg\min_W ||V - W \circledast A^{[l-1]}||^2 \tag{3.19}$$

where $\circledast$ is the convolutional operator. To solve for W^0, we solve for the partial of our objective function J with respect to the weights using the chain-rule based formula from [13]:

$$\frac{\partial J(W, A^{[l-1]})}{\partial W_{f,c,m,n}} = \sum_{f,q,r} \frac{\partial J(W, A^{[l-1]})}{\partial A^{[l]}_{f,q,r}} \cdot \frac{\partial A^{[l]}_{f,q,r}}{\partial W_{f,c,m,n}} \tag{3.20}$$

$$= \sum_{q,r} G_{f,q,r} A^{[l-1]}_{c,q-m,r-n} \tag{3.21}$$

where $G_{f,q,r} = \frac{\partial J(W,A^{[l-1]})}{\partial A^{[l]}_{f,q,r}}$. Since $A^{[l]} = (W \circledast A^{[l-1]})$, we can say that $J(A^{[l]}) = ||V - A^{[l]}||^2$. Now, we can solve for $G_{f,q,r} = \frac{\partial J(A^{[l]})}{\partial A^{[l]}_{f,q,r}}$:

$$G_{f,q,r} = \frac{\partial J}{A^{[l]}} \cdot \frac{\partial A^{[l]}}{\partial A^{[l]}_{f,q,r}} \tag{3.22}$$

$$= -2(V - A^{[l]}) \frac{\partial A^{[l]}}{\partial A^{[l]}_{f,q,r}} \tag{3.23}$$

$$= -2(V - A^{[l]}) \delta_{f,q,r} \tag{3.24}$$

$$= -2(V_{f,q,r} - A^{[l]}_{f,q,r}) \tag{3.25}$$

where δ is the Kronecker delta. Substituting this back,

$$\frac{\partial J(W, A^{[l-1]})}{\partial W_{f,c,m,n}} = -2 \sum_{q,r} (V_{f,q,r} - A^{[l]}_{f,q,r}) A^{[l-1]}_{c,q-m,r-n} \tag{3.26}$$

We set the partial $\frac{\partial J(W,A^{[l-1]})}{\partial W_{f,c,m,n}}$ equal to 0 to solve for $W^0_{f,c,m,n}$:

$$0 = -2\sum_{q,r}(V_{f,q,r} - A^{[l]}_{f,q,r})A^{[l-1]}_{c,q-m,r-n} \tag{3.27}$$

$$= -2\sum_{q,r}(V_{f,q,r}A^{[l-1]}_{c,q-m,r-n}) + 2\sum_{q,r}(A^{[l]}_{f,q,r}A^{[l-1]}_{c,q-m,r-n}) \tag{3.28}$$

$$= -2\sum_{q,r}(V_{f,-q,-r}A^{[l-1]}_{c,m-q,n-r}) + 2\sum_{q,r}(A^{[l]}_{f,-q,-r}A^{[l-1]}_{c,m-q,n-r}) \tag{3.29}$$

$$= -2(QV \circledast A^{[l-1]})_{f,c,m,n} + 2(QA^{[l]} \circledast A^{[l-1]})_{f,c,m,n} \tag{3.30}$$

$$= -2(V \circledast A^{[l-1]})_{f,c,m,n} + 2(Q(W^0 \circledast A^{[l-1]}) \circledast A^{[l-1]})_{f,c,m,n} \tag{3.31}$$

where Q is a rotation matrix of 180° about the f-axis. In other words, Q flips the filter of the convolution. Then, we can rearrange Eq. 3.31 to say:

$$(QV \circledast A^{[l-1]})_{f,c,m,n} = (Q(W^0 \circledast A^{[l-1]}) \circledast A^{[l-1]})_{f,c,m,n} \tag{3.32}$$

$$= (QW^0 \circledast QA^{[l-1]}) \circledast A^{[l-1]})_{f,c,m,n} \tag{3.33}$$

$$= (QW^0 \circledast (QA^{[l-1]} \circledast A^{[l-1]}))_{f,c,m,n} \tag{3.34}$$

Note that Eq. 3.34 comes from the property that $\circledast$ is associative. Since cross-correlation is convolution with a flipped filter, we can say:

$$(QV \circledast A^{[l-1]})_{f,c,m,n} = (QW^0 \circledast (A^{[l-1]} \star A^{[l-1]}))_{f,c,m,n} \tag{3.35}$$

$$= (QW^0 \circledast R_A)_{f,c,m,n} \tag{3.36}$$

where $\star$ is the cross-correlation operator, and R_A is the auto-correlation matrix of $A^{[l-1]}$ cross-correlated with itself. We can write Eq. 3.36 using the cross-correlation operator entirely:

$$(V \star A^{[l-1]})_{f,c,m,n} = (W^0 \star R_A)_{f,c,m,n} \tag{3.37}$$

Another way to write Eq. 3.37 is to say:

$$V \star A^{[l-1]} = W^0 \star R_A \; \forall W^0_{f,c,m,n} \in W^0 \tag{3.38}$$

Suppose $Y := V \star A^{[l-1]}$. Then, we can transform the cross-correlation $Y = W^0 \star R_A$ to the frequency domain:

$$\mathcal{F}(Y) = \overline{\mathcal{F}(W^0)} \cdot \mathcal{F}(R_A) \tag{3.39}$$

where $\mathcal{F}(\cdot)$ denotes the discrete Fourier transform, $\overline{(\cdot)}$ denotes the complex conjugate, and the operator $\cdot$ denotes element-wise multiplication. We assume that W^0 is only real valued, so in the Fourier domain, we can say:

$$\overline{\mathcal{F}(W^0)} = \mathcal{F}(Y)\mathcal{F}(R_A)^{-1} \tag{3.40}$$

$$\mathcal{F}(W^0) = \overline{\mathcal{F}(Y)\mathcal{F}(R_A)^{-1}} \tag{3.41}$$

$$W^0 = \mathcal{F}^{-1}\left(\overline{\mathcal{F}(Y)\mathcal{F}(R_A)^{-1}}\right) \tag{3.42}$$

where $\mathcal{F}^{-1}$ is the inverse discrete Fourier transform.

3.3.2 Constrained least squares solution

Now, similar to as we did in Section , we modify W^0 to assign a single key k^* to a new value v^*, where both k^* and v^* are vectors. We want to choose W^1 to minimize the loss function $J(W, A^{[l-1]})$:

$$W^1 = \arg\min_W ||V - W \circledast A^{[l-1]}||^2 \tag{3.43}$$

$$\text{subject to } v^* = W^1 \circledast k^* \tag{3.44}$$

Note that we can expand the convolutional operator of $v^* = W \circledast k^*$ in terms of its elements as:

$$v_f^* = (W \circledast k^*)_f \tag{3.45}$$

$$= \sum_c \sum_{m,n=-1}^{1} k_c^* W_{f,c,m,n} \tag{3.46}$$

We can write the constrained least squares problem (Eq. 3.43 and Eq. 3.44) as a Lagrangian function:

$$J(W, \lambda) = ||V - W \circledast A^{[l-1]}||^2 + \lambda(v^* - W \circledast k^*) \tag{3.47}$$

where λ is a vector of Lagrange multipliers. To solve for W^1, we solve for the partial of our objective function:

$$\frac{\partial J}{\partial W_{f,c,m,n}} = \sum_{f,q,r} \frac{\partial J}{\partial A_{f,q,r}^{[l]}} \cdot \frac{\partial A_{f,q,r}^{[l]}}{\partial W_{f,c,m,n}} + \frac{\partial J}{\partial v^*} \cdot \frac{\partial v^*}{\partial v_f^*} \cdot \frac{\partial v_f^*}{\partial W_{f,c,m,n}} \tag{3.48}$$

$$= \sum_{q,r} G_{f,q,r} A_{c,q-m,r-n}^{[l-1]} + \lambda \delta_f k_c^* \tag{3.49}$$

$$= \sum_{q,r} G_{f,q,r} A_{c,q-m,r-n}^{[l-1]} + \lambda_f k_c^* \tag{3.50}$$

where $G_{f,q,r} = \frac{\partial J}{\partial A_{f,q,r}^{[l]}}$ and δ is the Kronecker delta. Note that we already solved for $G_{f,q,r}$ in Eq. 3.25. We substitute Eq. 3.36 back into Eq. 3.50 for the following simplified expression:

$$(V \star A^{[l-1]})_{f,c,m,n} = (W^1 \star R_A)_{f,c,m,n} + \lambda_f k_c^* \tag{3.51}$$

We can write this in compact matrix-vector form since Eq. 3.51 must be true for all elements of W^1:

$$V \star A^{[l-1]} = W^1 \star R_A + \lambda k^* \tag{3.52}$$

We can use Eq. 3.38 to say:

$$W^0 \star R_A = W^1 \star R_A + \lambda k^* \tag{3.53}$$

$$W^1 \star R_A = W^0 \star R_A - \lambda k^* \tag{3.54}$$

We convert Eq. 3.54 to the Fourier space:

$$\overline{\mathcal{F}(W^1)} \cdot \mathcal{F}(R_A) = \overline{\mathcal{F}(W^0)} \cdot \mathcal{F}(R_A) - \mathcal{F}(\lambda k^*) \tag{3.55}$$

$$\overline{\mathcal{F}(W^1)} = \overline{\mathcal{F}(W^0)} - \mathcal{F}(\lambda k^*) \cdot \mathcal{F}(R_A)^{-1} \tag{3.56}$$

$$W^1 = W^0 - \mathcal{F}^{-1}(\overline{\mathcal{F}(\lambda k^*) \cdot \mathcal{F}(R_A)^{-1}}) \tag{3.57}$$

We have now expressed our rule edit as an update matrix, $\mathcal{F}^{-1}(\overline{\mathcal{F}(\lambda k^*) \cdot \mathcal{F}(R_A)^{-1}})$, to the pre-trained weights W^0.

3.4 Discussion

Much of what is presented in this chapter is experimental, and is still under work. In future iterations, we will experiment with this new update based on convolutional interactions, in hopes that the new update rule successfully captures the interactions between adjacent pixels such that we can edit in the inverse direction.

Chapter 4

Adding a New Class

Suppose we want to extend a classifier to classify new classes. Traditionally, this would require us to find new training data for the new classes, and either re-train an existing network or train a new network from scratch. In both cases, there is an upfront cost to retrieve large amounts of training data and a need for both time and computational resources to train the network. In this section, we experiment through various training strategies to add a new class to a pre-trained network that substantially reduces the need for new training images, for computational resources to train the network, and domain expertise about networks.

4.0.1 Summary of Results

In this chapter, we first present the idea of a class-specific filter, in which we train a new filter to detect the defining object parts of a desired class to add. We then activate only this filter in the last layer of the network to predict the new class. Then, we formulate baseline experiments to compare the performance of our class-specific filter, and conclude with an analysis. A highlight of our results are as follows:

- We successfully introduce a new class to a pre-trained network, using only one new training image and re-training only a single layer of the network.
- We use valuable human knowledge about our new class to design a class-specific filter, including teaching the filter what is *not* the class.

4.1 Class-Specific Filter

Suppose we want to add a new class to the VGG16 model trained on the MIT Places365 dataset: centaurs. Centaurs are mythological creatures that are half-man and half-horse. We choose to add the centaur class for two reasons. One, the centaur class is out-of-distribution in the sense that the centaur is not a scene category in the original MIT Places365 dataset. We can ensure that the model has never seen centaur images before. Second, the centaur is by definition a composition of two distinct object parts − a human upper body and a horse lower body − both of which *are* currently found in images the model is trained on. We can utilize this observation to see if it is sufficient for the network to have previously seen defining parts of the class, but not the class itself.

Fig. 4-1 shows examples of a variety of centaur images, along with how the pre-trained VGG16 model classifies these centaur images. By default, since we the pre-trained model does not contain the centaur class, none of these images can be (or are) classified as centaur. Our goal is to edit the classifier such that it does classify these images as "centaur." Notice that a predominant number of these centaur images are classified under the classes "museum-indoor" and "desert-sand", likely since the background of these centaur images are often associated with these classes.

In a traditional CNN, multiple filters might activate to detect a single class with each filter correlating with a visual concept. For example, Fig. 4-2 shows the top visual concepts for the centaur image and the filters that most correlate with each concept. A combination of activating these top visual concepts and their correlating filters results in the top predictive classes. In our class-specific filter CNN, our aim is to train a single additional filter to detect the defining object parts of the desired class and to activate this filter alone to predict the desired class. Fig. 4-2 shows that to predict the new centaur class, we train a new centaur-specific filter to detect the silhouette of the centaur. To classify the image as centaur, only the centaur-specific filter is activated.

In order to classify a new class, we must modify the network in two locations:

Figure 4-1: Classifications of selected centaur images. Since the network has no class for centaurs, none of the images are classified correctly. With our class-specific filter, our goal is to classify these images — and only these images — as "centaur."

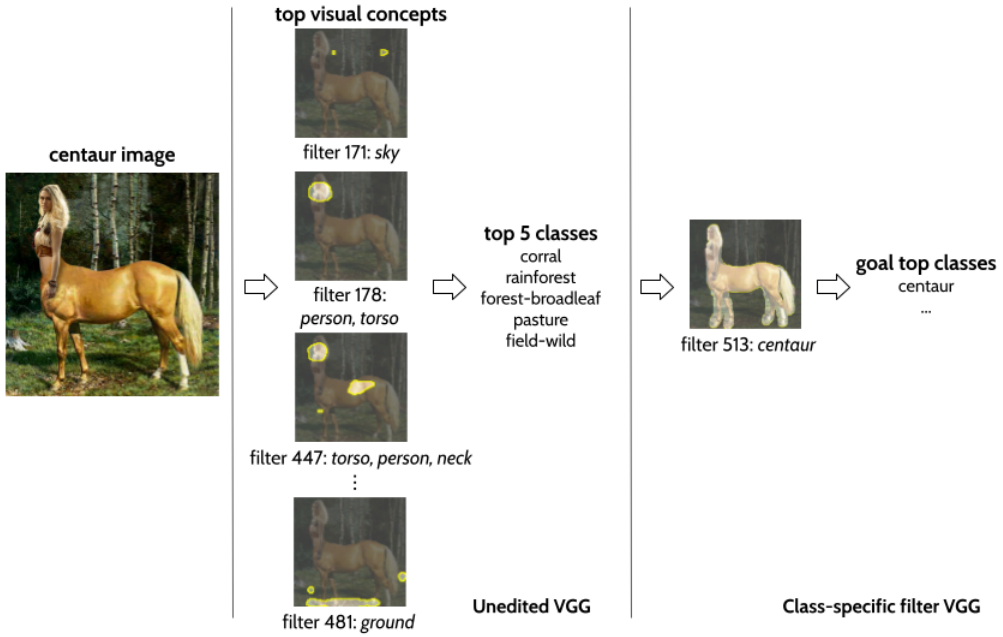

Figure 4-2: Prior to editing, the unedited network activates existing filters, which align with concepts, for the centaur image. Our goal in introducing the class-specific filter is for maximal activation of the the centaur filter for the centaur image, and none for non-centaur images.

one, introduce a class-specific filter at a desired target convolutional layer, and two, introduce an additional logit node that corresponds to the new class with an appropriate set of weights in the final softmax layer. Note that since we are only adding a class, and are not revising or removing existing classes, all other components of the network remain the same.

4.1.1 Optimizing the objective

First, we derive the formulation of training the class-specific filter. In principle, the ideal filter maps the defining features of a positive example of the class to a high weight value, and maps the remaining irrelevant features - for now, all the other features - to zero. We formulate finding the weights of the class-specific filter as a minimization problem where we use a single positive example of the class to define the relevant features, and multiple negative examples of what is *not* the class to define irrelevant features. The negative examples of what the class is not helps force the filter to not overfit to activate for other classes that may have similar defining features to the class we are adding. Suppose we have a convolutional layer l such that:

$$A_0^{[l]} = (W^0 * A_0^{[l-1]}) + B^0 \tag{4.1}$$

where W^0 denotes the original weights of the layer, $A_0^{[l]}$ denotes the output features computed by layer l, $A_0^{[l-1]}$ denotes the input features computed by the previous $l-1$ layers, and B^0 denotes the bias term. Note that the dimensions of W^0 are $f \times c \times m \times n$, where f is the filter number, c is the channel number, and $m \times n$ are the dimensions of the kernel. Also note that the dimensions of the input and output feature maps are $f \times q \times r$.

We wish to add a single filter at convolutional layer l such that:

$$A_1^{[l]} = (W^1 * A_0^{[l-1]}) + B^0 \tag{4.2}$$

where W^1 denotes the new weights of the layer, and $A_1^{[l]}$ denotes the new output

features computed by layer l. Note that the dimensions of W^1 are $(f+1) \times c \times m \times n$, and are the weights of a single convolutional layer. Also note that while the dimensions of the input feature map remains the same, the output feature map has dimensions $(f+1) \times q \times r$.

We want to find W^1 to minimize the loss function $J(W)$:

$$W^1 = \arg\min_W J(W) \tag{4.3}$$

$$= \arg\min_W \left[BCE(W * X^{[p]} || Y^{[p]}) + \lambda \sum_{i=1}^{N} BCE(W * X_i^{[n]} || Y^{[n]}) \right] \tag{4.4}$$

where $BCE(\cdot)$ denotes binary cross entropy loss, $X^{[p]}$ denotes the input representation of the positive example to the layer, $X^{[n]}$ denotes the input representation of a negative example to the layer, $Y^{[p]}$ denotes the target representation of the positive example, $Y^{[n]}$ denotes the target representation of the negative example, λ denotes a scaling factor applied to the loss from negative examples, and N denotes the number of negative examples.

4.1.2 Evaluation

To demonstrate this concept, we edit a pretrained VGG-16 network to add a centaur class to the MIT Places Dataset, as introduced in Fig. 4-2. First, we reason through our design of the centaur-specific filter. Then, we present a set of baseline experiments to evaluate the performance of our techniques, and finally a quantitative analysis of the centaur-specific filter in comparison to our baseline experiments.

4.1.3 Design of the class-specific filter

Intuitively, for maximal activation of the centaur shape, we design our centaur-specific filter to mimic the shape in pixel space of the centaur silhouette. We use gradient descent to minimize the objective. Fig. 4-3 shows our negative examples, which are the five top-scoring false positives that the edited network classifies as centaur, but are in fact of a different class, when we perform this optimization with the positive

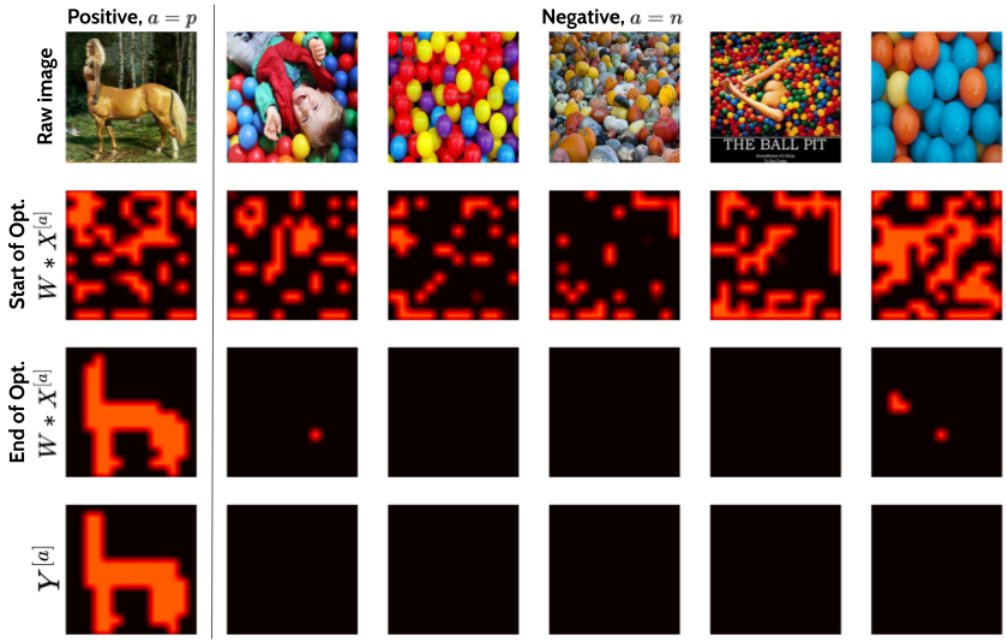

Figure 4-3: We visualize the feature maps of $W * X^{[a]}$ at the beginning and end of optimization, such that the training examples mimic the target $Y^{[a]}$. We also use negative examples to teach the filter what is *not* a centaur.

example only. Fig. 4-3 also shows that since we begin the optimization with a random set of weights, the feature maps $W * X^{[p]}$ and $W * X^{[n]}$ at the start of optimization are also random. Towards the end of weight optimization, as desired, we see that the feature map $W * X^{[p]}$ converges to the target $X^{[p]}$, which is the silhouette of the centaur, and that the feature maps $W * X_i^{[n]}$ converges to the target $X^{[n]}$, which is zero.

4.1.4 Baseline experiments

Since the original network has no centaur class, the final softmax layer does not contain weights for the new centaur class. With the centaur-specific filter, the final softmax layer weights will be a one-hot vector, with activation only for the centaur filter. Naturally, it begs the questions: without the class-specific filter, what might our baseline final layer weights be? And eventually, what might it say about our ability to classify centaurs with the class-specific filter relative to our baseline? We

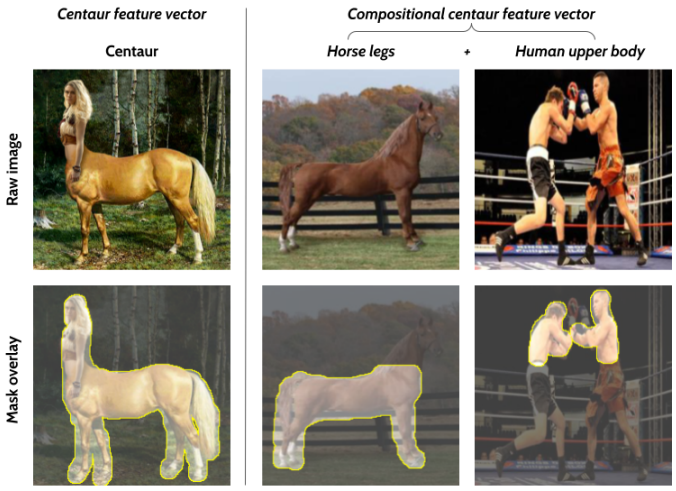

Figure 4-4: Since the original network has no centaur class, the final softmax layer does not contain weights for the new centaur class, we formulate baseline experiments where we select weights for the centaur class in the final layer. Here, the centaur feature vector is the feature vector for the masked centaur shape and the compositional feature vector is the feature vector for horse legs combined with that of the human upper body.

propose three simple baseline experiments for comparison, all of which determine the set of weights in the final softmax layer for the centaur class:

Random vector. We choose a vector with random weights sampled from a standard normal distribution.

Class feature vector. We choose the normalized feature vector that has context matched to the new class object at a later convolutional layer. In this example with the centaur, we find the feature vector for the centaur object using the manual mask of the entire centaur shape in the centaur image shown in Fig. 4-4. Note that since this image is out-of-distribution, it is likely the network has no robust feature vector for the centaur.

Compositional class feature vector. We combine the feature vectors that have context matched to the defining features of the new class object. In this example with the centaur, we might describe a centaur as a human face and chest along with a horse body, legs, and tail. As shown in Fig. 4-4, we find the feature vectors for each component from a single image belonging to a class the network is trained on, add them in feature space, and normalize the final vector to produce a single compositional

centaur feature vector.

Our goal with adding a new class is two-fold: one, we wish to maximize the true positive rate, in other words classify all the centaurs as centaurs; and two, to minimize the false positive rate, in other words, to not classify the remaining images as centaurs. In other words, we wish to minimize the collateral effect on the original classes since we wish for the original classifications to remain the same. To measure the relationship between true positive and false positive rates, we use the area under the receiver-operator curve (ROC), where the positive class is the newly added class and the negative class is all of the remaining classes combined. Since the positive and negative class sizes are imbalanced, we allow each sample to provide a weighted contribution, relative to class size, to the overall score. Note that a higher ROC score indicates more correct predictions. By nature of our goal to add a new class, there exists no prior validation dataset for this new class. To build a validation dataset for evaluation, we scrape Google Search for sixty centaur images, shown in Fig. 4-1, and manually verify that these are indeed centaurs.

Fig. 4-5 motivates the promise behind a class-specific filter; the relative mediocre performance (i.e. $ROC \approx 0.6$) of both the centaur feature vector and the compositional centaur feature suggest that the later layer filters in the network are not greatly associated with centaurs or a combination of the components of centaurs. Instead, we need to build a filter tuned to detect a centaur to be able to classify it well. For future experiments, we use the highest scoring baseline method - the compositional feature vector - as our primary baseline comparison.

4.1.5 Analysis of the class-specific filter

In this section, we add the centaur-specific filter to the network along with final layer weights for the centaur class as a one-hot vector, activated only for the newly added centaur-specific filter. Notice that in our filter optimization objective, Eq. 4.4, we use λ as a scaling factor to regulate the contribution of the negative examples to the overall loss. Fig. 4-6 shows the ROC curves and loss over optimization steps for an experiment that changes λ from 0 (i.e. no negative examples used to optimize the

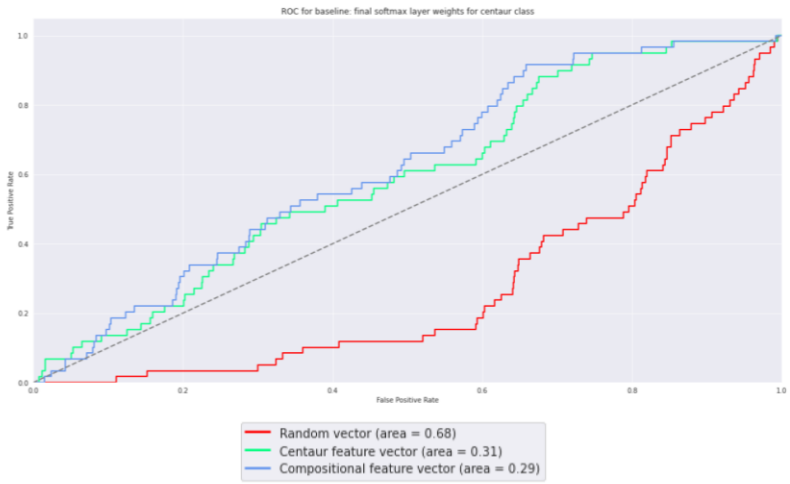

Figure 4-5: Receiver-operator curves for the baseline experiments. The compositional feature vector performs better than the others on almost all thresholds, so we use this as our baseline going forward.

filter) to 0.05 using the top five false positive images shown in Fig. 4-3. Compared to the baseline of the compositional feature vector with $ROC = 0.54$, we see a substantial increase to $ROC = 0.82$ with the class-specific filter at $\lambda = 0$. In fact, the centaur-specific filter for all λ values are higher than the baseline, demonstrating the utility of our approach in using a class-specific filter to edit an existing network to include a new class.

Fig. 4-7 looks at the classification of the validation dataset with the class-specific filter model, which is the set of images seen earlier in Fig. 4-2. Out of the 59 centaur images, we are able to classify 24 images ($\approx 40\%$) correctly as centaur; of the remaining classified incorrectly, $\approx 80\%$ are classified in the same top false positive classes seen earlier in Fig. 4-1. However, notice that these false positive images tend to be renderings without contextual background (e.g. a white background) or clip art. It is possible that the model relies on contextual scene information, such as nature in the background, and/or that the model is unable to process clip art well, if having never seen clip art images before. The relatively high ROC score of our edited classifier suggests that the class-specific filter edited classifier minimizes false negatives; Fig. A-1 in the Appendix shows that a naively edited model with a low

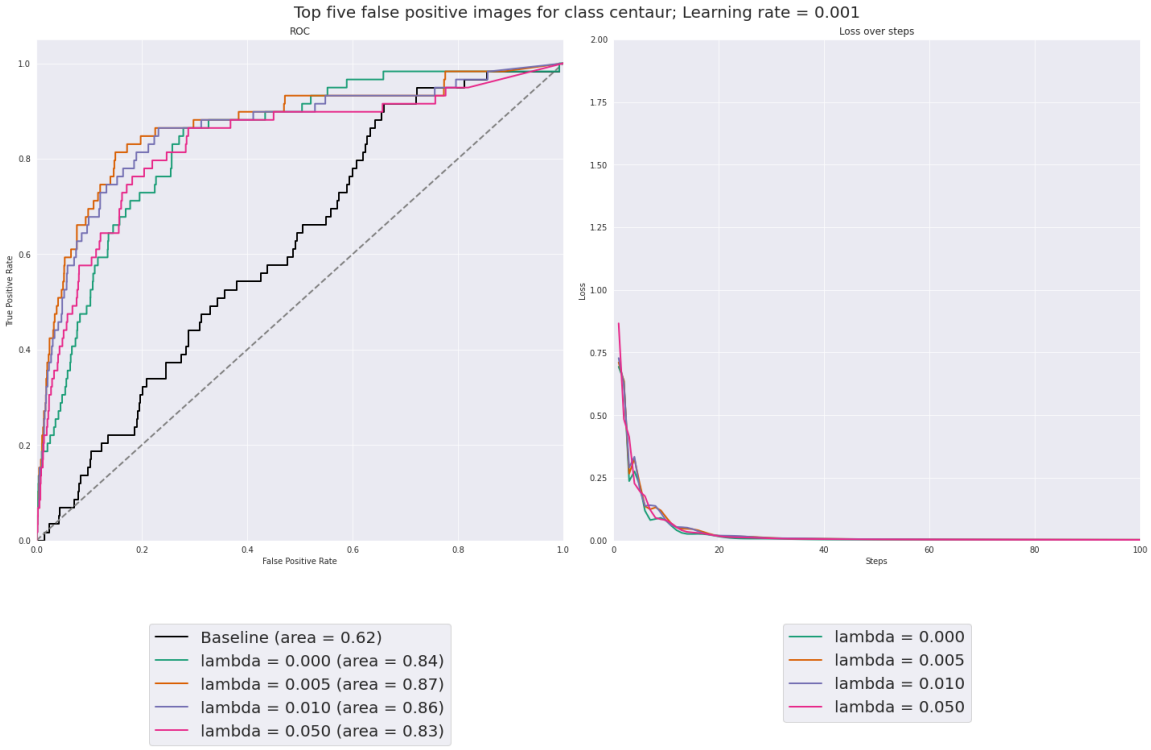

Figure 4-6: Shows the ROC curves and loss over optimization steps for an experiment that changes λ from 0 (i.e. no negative examples used to optimize the filter) to 0.05 using the top five false positive images shown in Fig. 4-3.

Figure 4-7: Classification of the validation dataset with the class-specific filter model, which is the set of images seen earlier in Fig. 4-1.

ROC score incorrectly classifies almost all images as centaur.

In addition to seeing a substantial performance increase with the centaur-specific filter in Fig. 4-6, we also see that the inclusion of negative examples does improve how well we predict centaurs. Just as important as teaching the network what *is* a centaur, the intention behind using negative examples is to help teach the network what is *not* a centaur. The question now is: what other negative examples might we use and how many should be used? Fig. A-2 in the Appendix shows the performance of the class-specific filter when using random images sampled from the MIT Places validation dataset as negative examples.

4.2 Discussion

With minimal, but powerful intervention to the network, we are able to successfully introduce a new class to a pre-trained network, forgoing the need for more than one

new training image or the need to re-train the network itself.

Optimal filter. In our optimization of the filter, Eq. 4.4, we intuitively choose our target $Y^{[p]}$ to be the centaur silhouette. While the centaur silhouette is an informative and promising target, it may not capture the most valuable information: what makes a centaur unique? At least to reduce the false positive predictions, it is useful to find the parts or features of the class that are unusual or distinctive to the class. Moreover, careful design of the target provides an important opportunity for human input to intervene with the network to adjust what the network currently says. Other possibilities for the target include: a mask of the unique parts of the centaur, e.g. the location at which the human part transforms into the horse part, or a max pooling of the centaur shape.

Separate optimization for final layer weights. In our current implementation, we assume that we only need the centaur-specific filter to detect centaurs. Instead, it is likely that including existing filters, such as ones chosen by human input that correlate with desirable visual concepts or ones chosen by the network through another optimization of the final layer weights, improves classification ability.

Choice of negative examples. Certainly the type of negative examples, and likely the number of negative examples greatly affects the performance of the filter. More experiments need to be conducted to better understand the role and effect of negative examples in this optimization.

Limitations to our technique. At present, our technique is a type of one-shot learning; in other words, we still require a single example of the class from which we build the target filter. While we consider human input in the form of designing the target filter to be undeniably valuable, there is certainly a cost to this intervention. Although our current method only requires human input to mask the relevant features of the single class image for a working class-specific filter, users might find that more subtle design choices must be made to find the optimal class-specific filter. Certainly future iterations have the potential to automate these human decisions further.

Chapter 5

Conclusion

In this thesis, we have shown how to update a classifier to our desire using basic intuitive knowledge about how humans classify the world. Doing so allows us to reduce the reliance on computational resources, new training images, and domain knowledge. We began by examining the forward direction of rule rewriting. We explore how to use the rewriting technique to expand a class to unseen scenarios, how to identify spurious correlations in our model, and how to use rewriting to significantly reduce the effects of a spurious correlation.

We explore the two fundamental ideas to rewriting, namely that we can think of our layer's weights as a memory bank that stores and associates two feature representations together and that visual concepts align with these feature representations. As a result, we can use human knowledge about visual concepts to directly formulate the feature representations to edit.

Next, we explore another interpretation of the rewriting technique, the inverse direction. The inverse direction poses unique challenges with computational resources, but is a promising avenue to update rules specifically in the classifier setting. Finally, we conclude with how to use the fundamental ideas discussed prior to add an entirely new class to our model.

5.1 Related work

In this section, we highlight related work and how it compare to our rewriting technique:

Transfer learning. Transfer learning transfers knowledge from one domain to another domain [31]; in practice, domain adaptation adapts the source domain to reduce the difference between domains, one-shot or few-shot learning generalizes using only a few samples [27], meta-learning trains on a variety of learning tasks, such that it can solve new learning tasks using only a small number of training samples [8], and fine-tuning updates model parameters to a specific set of images [17]. All of these techniques require substantial training time with additional annotated data, or extend classification to a particular set of images. Our rewriting technique only requires re-training a single layer of the network and directly modifies the rules of the network, not the pre-trained images, to extend classification.

Identifying and resolving spurious correlations. [26] designs a reward function that incentivizes an agent to do an intervention to find errors in the causal model. [10] manipulates the objective function to learn a disentangled representation. [2] proposes variations of auto-encoders that lead to better disentanglement. Similar to the techniques in transfer learning, ideas in this space focus on revising the optimizer to generate new models, as opposed to post-hoc changes to the pre-trained model.

Appendix A

Figures

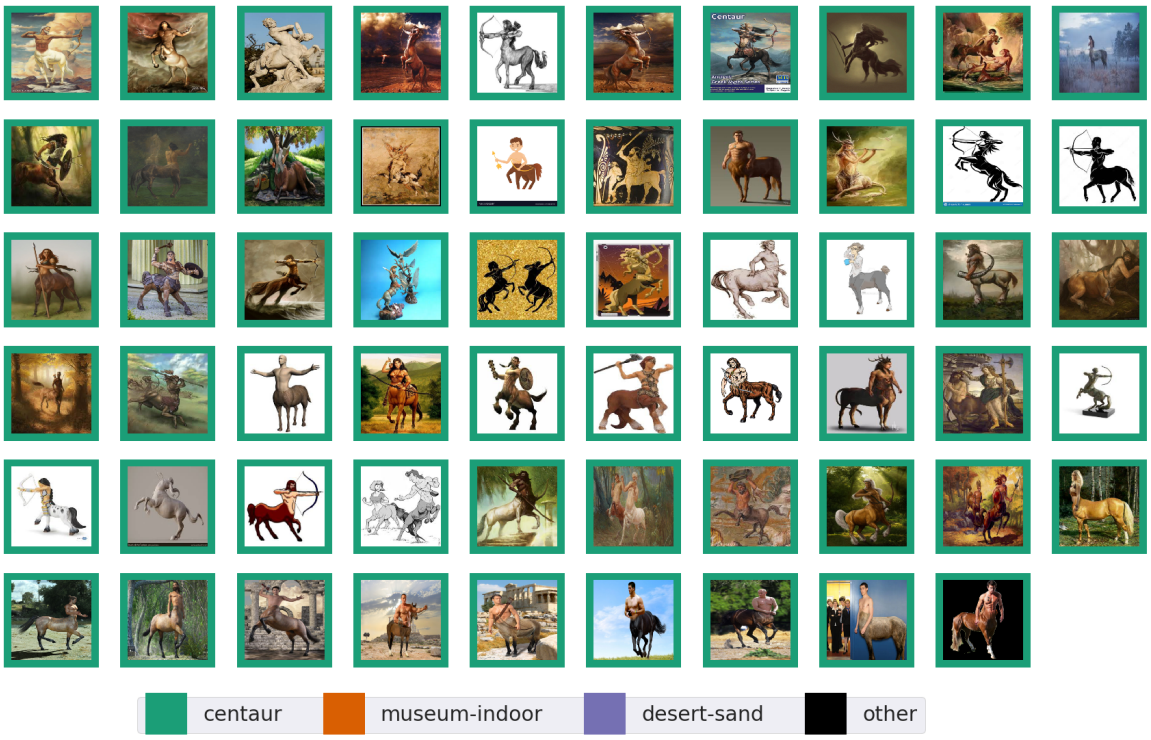

Figure A-1: Classifications produced by an edited model that uses no class-specific filter and uses the compositional centaur feature vector as the centaur class's final layer weights, incorrectly classifies almost all images as centaur.

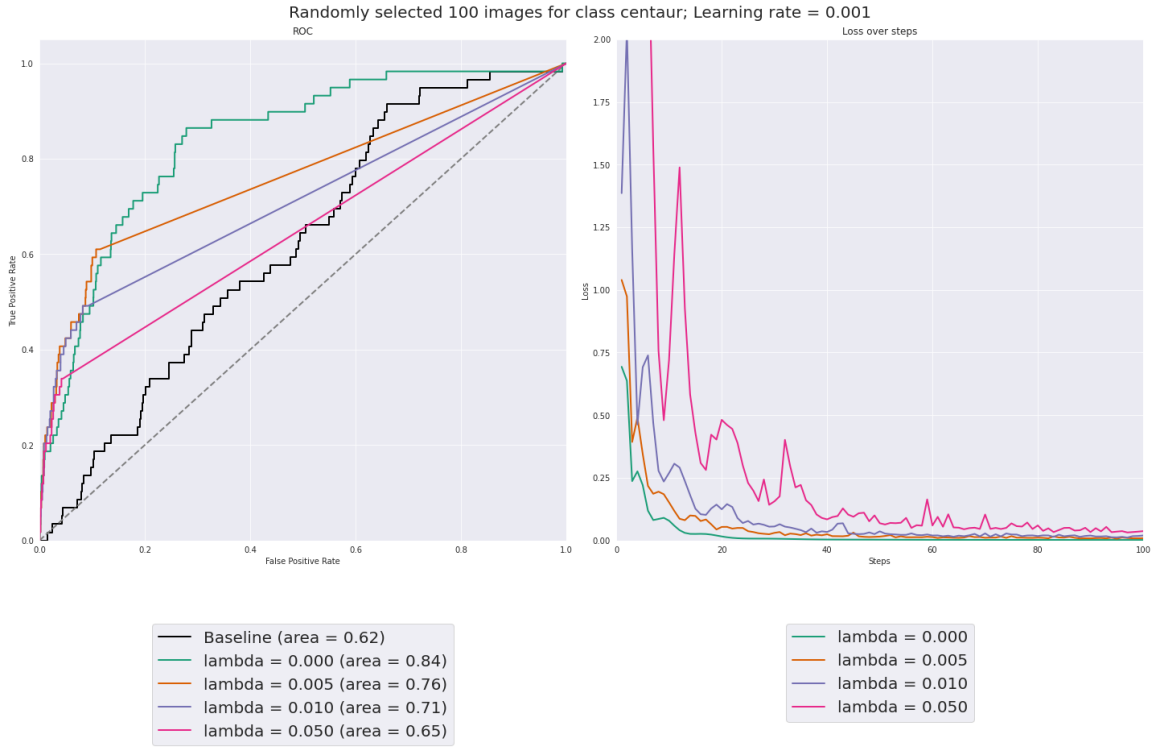

Figure A-2: Shows the ROC curves and loss over optimization steps for an experiment that changes λ from 0 (i.e. no negative examples used to optimize the filter) to 0.05 using 100 randomly selected non-centaur images from the validation dataset.

Bibliography

[1] David Bau, Steven Liu, Tongzhou Wang, Jun-Yan Zhu, and Antonio Torralba. Rewriting a deep generative model, 2020.

[2] Christopher P. Burgess, Irina Higgins, Arka Pal, Loic Matthey, Nick Watters, Guillaume Desjardins, and Alexander Lerchner. Understanding disentangling in -vae, 2018.

[3] Alexandra Chouldechova. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments, 2017.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[5] Finale Doshi-Velez and Been Kim. Towards A Rigorous Science of Interpretable Machine Learning. *arXiv:1702.08608 [cs, stat]*, March 2017. arXiv: 1702.08608.

[6] Marat Dukhan. The indirect convolution algorithm, 2019.

[7] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Brandon Tran, and Aleksander Madry. Adversarial robustness as a prior for learned representations, 2019.

[8] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[9] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, 2015.

[10] Benoit Gaujac, Ilya Feige, and David Barber. Learning disentangled representations with the wasserstein autoencoder, 2020.

[11] Marzyeh Ghassemi, Tristan Naumann, Peter Schulam, Andrew L. Beam, Irene Y. Chen, and Rajesh Ranganath. A review of challenges and opportunities in machine learning for health, 2018.

[12] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining Explanations: An Overview of Interpretability of Machine Learning. *arXiv:1806.00069 [cs, stat]*, February 2019. arXiv: 1806.00069.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Bryce Goodman and Seth Flaxman. European union regulations on algorithmic decision-making and a "right to explanation". 2016.

[15] Boris Hanin and David Rolnick. Complexity of linear regions in deep networks, 2019.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[17] Ahmadreza Jeddi, Mohammad Javad Shafiee, and Alexander Wong. A simple fine-tuning is all you need: Towards robust deep learning via adversarial fine-tuning, 2020.

[18] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, and Mingli Song. Neural style transfer: A review, 2017.

[19] Jon Kleinberg, Jens Ludwig, Sendhil Mullainathan, and Ashesh Rambachan. Algorithmic Fairness. *AEA Papers and Proceedings*, 108:22–27, May 2018.

[20] Zhizhong Li and Derek Hoiem. Learning without forgetting, 2016.

[21] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[23] Stephan Rabanser, Stephan Günnemann, and Zachary C. Lipton. Failing loudly: An empirical study of methods for detecting dataset shift, 2018.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[25] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey, 2017.

[26] Sergei Volodin, Nevan Wichers, and Jeremy Nixon. Resolving spurious correlations in causal models of environments via interventions. 2020.

[27] Yaqing Wang, Quanming Yao, James Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning, 2019.

[28] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric, 2018.

[29] Bolei Zhou, David Bau, Aude Oliva, and Antonio Torralba. Interpreting deep visual representations via network dissection, 2017.

[30] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[31] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning, 2019.