TU WIEN Informatics

Visual Analytics for Convolutional Neural Network Robustness

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Stefan Sietzen, BSc
Matrikelnummer 0372194

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr. Manuela Waldner
Mitwirkung: Dipl.-Ing. Mathias Lechner
Univ. Ass. Dr. Ramin Hasani

Wien, 11. Jänner 2022

Stefan Sietzen	Manuela Waldner

TU WIEN Informatics

Visual Analytics for Convolutional Neural Network Robustness

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Stefan Sietzen, BSc
Registration Number 0372194

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr. Manuela Waldner
Assistance: Dipl.-Ing. Mathias Lechner
Univ. Ass. Dr. Ramin Hasani

Vienna, 11th January, 2022

Stefan Sietzen	Manuela Waldner

Erklärung zur Verfassung der Arbeit

Stefan Sietzen, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Jänner 2022

Stefan Sietzen

Acknowledgements

I want to thank my supervisor Manuela Waldner for supporting me to freely explore my ideas, for her great advice and guidance on this thesis, and for her patience.

I also want to thank my parents, my partner, and my friends for their support during my studies.

The work presented in this thesis has been partially described in our paper "Interactive Analysis of CNN Robustness", published in *Computer Graphics Forum (Proceedings of Pacific Graphics 2021)*, 40(7), 2021. I want to thank my co-authors Mathias Lechner, Judy Borowski, Ramin Hasani, and Manuela Waldner for their valuable contribution.

Furthermore, I want to thank Chris Olah and Nick Cammarata for their encouragement and feedback on an early prototype of the presented application through the Distill.pub Slack workspace.

Kurzfassung

Convolutional Neural Networks (CNNs) sind ein Typ von Machine Learning Modellen, der weit verbreitet ist bei Computer Vision Systemen. Trotz ihrer hohen Genauigkeit ist die Robustheit von CNNs oft schwach. Ein für Bildklassifizierung trainiertes Modell könnte beispielsweise ein Bild falsch klassifizieren nachdem das Bild leicht gedreht wurde, bei leichter Unschärfe, oder bei veränderter Farbsättigung. Außerdem sind CNNs anfällig gegen sogenannte "Adversarial Attacks", Methoden, um analytisch minimale Veränderungen am Bild zu generieren. Diese sind für den Menschen nicht wahrnehmbar, können das Klassifizierungsmodell aber in die Irre führen. Es wurden verschiedene Trainingsmethoden entwickelt, um die Robustheit von CNNs zu verbessern.

In dieser Arbeit untersuchen wir die Robustheit von CNNs mit zwei Ansätzen: Zuerst visualisieren wir Unterschiede zwischen standard und robusten Trainingsmethoden. Dafür verwenden wir Feature Visualization - eine Methode zur Visualisierung von Mustern, auf die individuelle Neuronen eines CNNs ansprechen. Darauf aufbauend stellen wir eine interaktive Visualisierungsanwendung vor, die die Nutzer eine 3d Szene manipulieren lässt, während sie gleichzeitig die Vorhersagen sowie Aktivierungen aus den versteckten Ebenen des CNNs beobachten können. Um standard und robust trainierte Modelle vergleichen zu können, erlaubt die Anwendung die gleichzeitige Beobachtung von zwei Modellen. Um die Nützlichkeit unserer Anwendung zu testen, führten wir fünf Case Studies mit Machine Learning Experten durch. Im Zuge dieser Case Studies und unserer eigenen Experimente konnten wir mehrere neue Erkenntnisse über robust trainierte Modelle gewinnen, von denen wir drei quantitativ verifizieren konnten. Trotz der Möglichkeit, zwei hochperformante CNNs in Echtzeit zu untersuchen, läuft unsere Anwendung clientseitig in einem standard Webbrowser und kann als statische Website übertragen werden, ohne einen performanten Backend-Server zu benötigen.

Abstract

Convolutional neural networks (CNNs) are a type of machine learning model that is widely used for computer vision tasks. Despite their high performance, the robustness of CNNs is often weak. A model trained for image classification might misclassify an image when it is slightly rotated, blurred, or after a change in color saturation. Moreover, CNNs are vulnerable to so-called "adversarial attacks", methods where analytically computed perturbations are generated which fool the classifier despite being imperceptible by humans. Various training methods have been designed to increase robustness in CNNs.

In this thesis, we investigate CNN robustness with two approaches: First, we visualize differences between standard and robust training methods. For this, we use feature visualization, a method to visualize the patterns which individual units of a CNN respond to. Subsequently, we present an interactive visual analytics application which lets the user manipulate a 3d scene while simultaneously observing a CNN's prediction, as well as intermediate neuron activations. To be able to compare standard and robustly trained models, the application allows simultaneously observing two models. To test the usefulness of our application, we conducted five case studies with machine learning experts. During these case studies and our own experiments, several novel insights about robustly trained models were made, three of which we verified quantitatively. Despite its ability to probe two high performing CNNs in real-time, our tool fully runs client-side in a standard web-browser and can be served as a static website, without requiring a powerful backend server.

Contents

CHAPTER 1

Introduction

The undeniable success of deep learning (DL) in recent years has led to the adoption of DL-based methods for many technical applications. One prime example is computer vision (CV), where convolutional neural networks (CNNs) sometimes achieve super-human accuracy [HZRS15]. These impressive results rely on an identical distribution of the training- and a (held-out) test dataset, but often the generalization to perturbed data is weak. Such perturbations can be simple color shifts, blur, additive noise, or image transformations that are different from those used in training, but also more sophisticated procedures such as so-called adversarial examples. These are images where the CNN's gradient of the class probabilities with respect to the input pixels is used to compute minimal, often humanly-imperceptible perturbations that fool the network into making a false prediction (Example shown in Figure 1.1).

This apparent misalignment between human and CNN vision has led researchers to investigate and improve the robustness of machine learning (ML) models. Robustness is of central importance for safety-critical use cases, such as medical applications or self-driving cars [LHA+20]. A model that makes its predictions based on highly different features compared to humans can not be trusted, therefore researchers have worked on improving robustness against a variety of perturbations by incorporating perturbed examples into the training procedure [GSS15]. These robustly trained models often behave differently from their standard-trained counterparts when faced with other perturbations than they have been trained on, as we show in Section 7.1.

To gain a better understanding of which features are learned by CNNs, researchers have conducted experiments directly comparing human perception against CNN perception [GRM+18], which showed a bias of standard-trained CNNs towards texture versus shape and a lack of understanding of global structure. Others have shown that standard-trained CNNs are highly sensitive to high frequency perturbations [YLS+19] or have even indicated that CNNs mostly rely on surface statistical regularities [JB17].

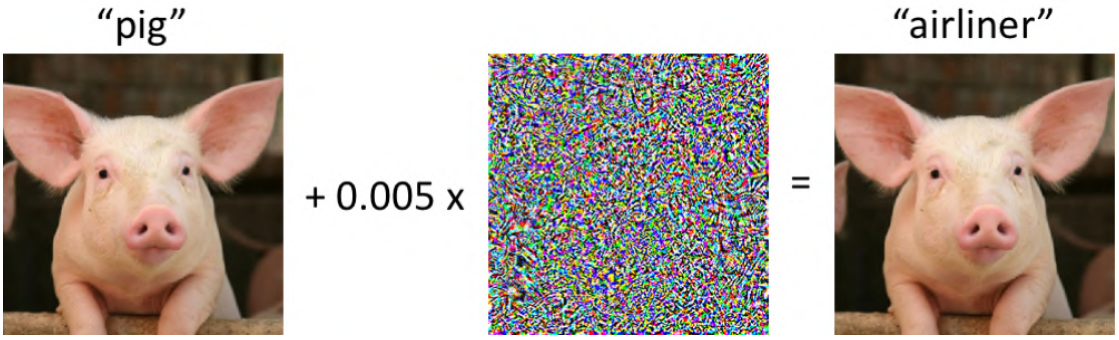


Figure 1.1: Illustration of an adversarial attack, taken from a *gradientscience.org* blogpost [MS18]: The left image is correctly classified as “pig“. Adding the middle image, multiplied by 0.005, results in the right image, which is incorrectly classified as “airliner“, while having no human-visible difference to the original.

The exaggerated reliance on high-frequency features is strongly linked to the existence of adversarial examples. While human vision is able to reason about global structure and therefore can not be fooled by local high-frequency changes, standard CNN predictions are brittle. They can be easily fooled by noise-like perturbations without making structural changes to the input image.

Adversarial examples have sparked a large amount of research recently. A multitude of attacks and defenses have been developed, overpowering each other in quick succession. In 2019, Madry et al. [MMS+18] have theoretically shown that projected gradient descent (PGD) is the strongest first-order attack. They conjectured that incorporating PGD-generated adversarial examples into the training procedure provides provable robustness against all first-order attacks of a certain perturbation magnitude. This is called *adversarial training*. Interestingly, strong adversarial examples targeting adversarially trained models exhibit a visible low-frequency structural change when compared to the original image. This is a highly different behaviour than found when attacking standard-trained models (example shown in Figure 1.2), and indicates that adversarial training forces the model to shift its focus more towards low-frequency features compared to standard training.

To visualize what features a CNN has learned, various feature visualization techniques have been developed. These techniques differ in the way they generate and regularize the respective visualization. Regularization techniques are used to visually approximate natural images but often come with the cost of constraining the space of possible visualizations and therefore hiding properties of the learned features.

Although feature visualization techniques represent a powerful lens into CNNs, they have been mostly used to analyze standard trained models rather than investigating models specifically optimized for some type of robustness. This leads us to our first hypothesis:

H1: Feature visualization can help to understand the difference of learned

Figure 1.2: Left: Original image, classified as "Malinois" by both standard and adversarially trained models. Center: Image after 12 iterations of epsilon 100.0 L_2-bounded PGD attack on std.-trained model towards "African elephant". Right: Image after same attack on adversarially trained model.

features between standard and robust training.

Thus, our first contribution is to shed light onto the development of learned features during standard training and during two robust training variants by using feature visualization techniques. The training variants are adversarial training [MMS+18] and training on the *Stylized ImageNet* (SIN) dataset [GRM+18]. We generate feature visualizations for various checkpoints along the training process.

As the focus of this work is the comparative investigation of learned features between differently trained models, we need to use visualizations that are able to uncover all subtleties of learned features. We therefore use iterative activation maximization [EBCV09] with simple transformation robustness [OMS17]. This avoids checkerboard artifacts and provides some degree of frequency smoothing. Also, we visualize variants with additional color- and frequency decorrelation, leading to more natural apperance.

Our second hypothesis is inspired by the various experiments on CNN robustness we encountered in literature, which mostly focus on a single type of perturbation. These experiments often take hours to set up according to domain experts who participated in our case study.

H2: Interactively applying and combining a large palette of image perturbations while simultaneously inspecting the responses of a single or multiple CNNs can help to investigate the CNNs' robustness.

To test this hypothesis, we first developed an interactive visual analytics application called *Perturber* with above described capabilities. Perturber allows the manipulation of a rendered 3d scene by camera transformations, texture and lighting changes, as well as object morphing and background changes. The resulting image can be further manipulated by various post-processing effects such as hue-, saturation- or brightness shifts, frequency-based effects such as blurring, and even by generating adversarial

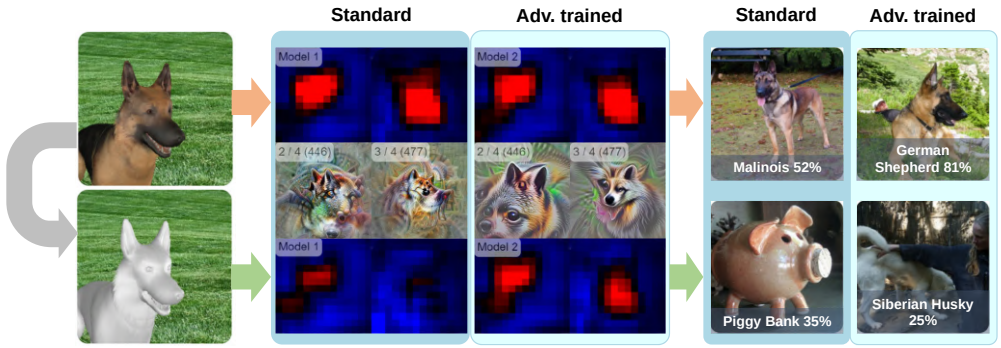


Figure 1.3: Example of an interactive exploration in Perturber. Left: A rendered image of a German Shepherd is perturbed by removing its texture. Center: After the texture has been removed, the activations for two selected "dog-relevant" units can be seen to decrease heaviliy in the standard model, while they decrease only slightly in the adversarially trained model. "The dog-relevant" units are visually identified by their respective feature visualizations. Right: For making the class prediction, the standard model seems to predominantly rely on texture, and therefore classifies the smooth, untextured dog as a piggy bank. The adversarially trained model apparently relies more on color and shape, leading to the perturbed image being classified as a Siberian Husky. Figure taken from [SLB+21].

attacks. The final image then serves as input to one or two CNNs whose activations are visualized interactively, i.e. each input change immediately causes a synchronously updated activation visualization at interactive framerates. The application visualizes intermediate featuremap activations as heatmaps with negative activations depicted in blue color and positive activations depicted in red color. Class probabilities are visualized as a top-5 bar chart and by dataset examples from the respective class. Perturber and its design reasoning are described in more detail in Chapter 5, the general concept is depicted in Figure 1.3.

We then conducted case studies with five deep learning researchers as participants, two of which were involved in the design process by incorporating their early feedback. The case studies showed that Perturber helps users to quickly generate hypotheses about model vulnerabilities and to qualitatively compare model behavior. Using quantitative analyses, we could generalize participants' insights to other CNN architectures and input images than the ones used in Perturber, yielding new insights about the vulnerability of adversarially trained models.

In summary, our main contributions are:

- We extensively investigated standard and robust training by generating feature visualizations for finely-grained sequences of checkpoints, with a focus on comparison between the training methods.

- We developed Perturber, a visual analytics application that allows the user to interactively manipulate and perturb a 3d input scene, while observing intermediate responses and predictions of CNNs classifiers. The application allows comparing up to three models with identical architecture but different parameters.
- We conducted five expert case studies and evaluated the results. We collected quantitative evidence for three hypotheses that were generated by using Perturber.

CHAPTER 2

Machine Learning Background

In this chapter, we will introduce the most important concepts in the ML context necessary to read this thesis. We will start by looking at the basics of convolutional neural networks, the *ImageNet* dataset, and transfer learning in Section 2.1. Then we will look at some details of topics related to adversarial attacks in Section 2.2 before dealing with feature visualization in Section 2.3

2.1 Deep Learning Basics

Deep learning is a special branch of machine learning dealing with neural networks with multiple hidden layers. In contrast, early works on neural networks like the *Perceptron* algorithm [Ros58] only contained an input and an output layer, the parameters being coefficients of a linear hyperplane representing the decision boundary. Introducing hidden layers in combination with non-linear activation functions increases the representative power of a neural network. It can be shown that even a shallow neural network with one hidden layer and a non-linear activation function can be a universal function approximator [Cyb89]. Deep model architectures using multiple hidden layers help the hierarchical learning of features, as the early layers can learn lower level features which the later layers then can recombine to higher level features. The efficient training of so-called deep neural networks (DNNs) has been enabled by the development of the *backpropagation* algorithm [RHW85].

2.1.1 Convolutional Neural Networks

Convolutional neural networks are a type of neural network, which make heavy use of weight sharing between spatial locations within so-called convolutional layers. In contrast, vanilla neural networks make use of fully-connected layers, where each input neuron is connected to each output neuron by a uniquely-weighted connection. Convolutional

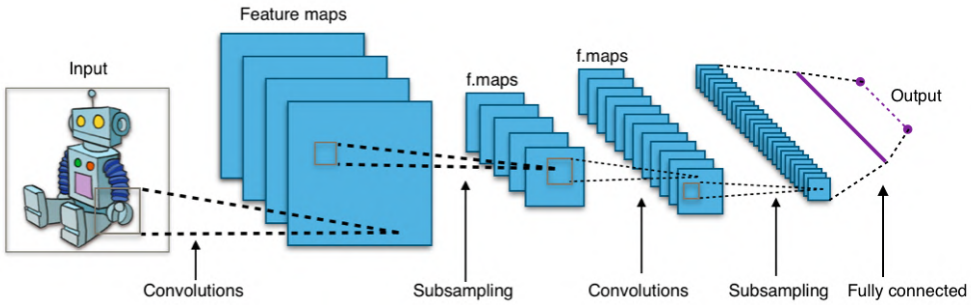


Figure 2.1: Schematic depiction of a CNN. The kernels of each layer (small squares) are shared across all feature map locations. Figure created by Wikipedia user Aphex34, CC BY-SA 4.0.

layers can take data with one, two, or three spatial dimensions as input, but are mostly known for their use with images (two spatial dimensions). The number of weights of a convolutional layer solely depends on the kernel size and the number of input and output channels, not on the spatial size of the image. A schematic depiction of a CNN can be seen in Figure 2.1. CNN layers learn a number of convolutional filters and output the same number of channels, one per filter, in the form of so-called feature maps. A standard CNN layer is represented by a kernel with $h \times w \times i \times o$ learned weights, where h is the kernel height, w is the kernel width, i is the number of input channels and o is the number of output channels, and an additional set of o bias values. Notably, a convolutional layer acting on spatially two-dimensional data like images, learns a number of three-dimensional filter kernels, consisting of the channel dimension in addition to the two spatial dimensions.

In order to reduce the spatial dimensionality of the information flow in CNNs, downsampling layers are being used. Commonly used downsampling layers are *max-pooling* and strided convolution. Max-pooling layers usually split the input image into multiple small patches and only output the maximum value from each patch. A typical max-pooling configuration would be a kernel size of 3 with a stride of 2. This would output an image half the width and height of the input image because of stride 2, with each pixel containing the maximum value from a 3×3 window of the input image. A similar configuration is typically used for strided convolutions. Here, the downsampling is performed by convolving the input with a learnable 3×3 kernel, just like in normal convolutional layers, but skipping every other spatial position resulting in an output size reduced by two in each dimension. In order to avoid reducing the dimensionality excessively, spatial downsampling layers are often accompanied by a doubling of the channel dimension. This can be seen in Figure 2.2, where each for each max pooling layer except for the last this applies. Note that doubling the number of channels while halfing both spatial dimensions still amounts to an overall halfing of the dimensionality, as the spatial halfing reduces dimensions by four.

The convolutional layers in a CNN are considered the feature extractors, which distill high

level semantic features into a lower dimensional space from a very high dimensional input. In the example of the *VGG 16* architecture (Figure 2.2), the input has $224 \times 224 \times 3$ dimensions, whereas the extracted feature vector has $7 \times 7 \times 512$ dimensions, a reduction by a factor of 6. This is a comparatively weak reduction, caused by the fact that VGG 16 keeps the 7×7 spatial dimensions before flattening the features and feeding them into the classification head, which consists of fully connected layers. In contrast, the *Inception V1* architecture incorporates a global average pooling layer that computes the mean of each of the final 1024 feature maps before the fully-connected classifier. This design choice decouples the spatial layout of the final features from the classifier and thus resolves the issue of a fixed input resolution that is mandatory otherwise.

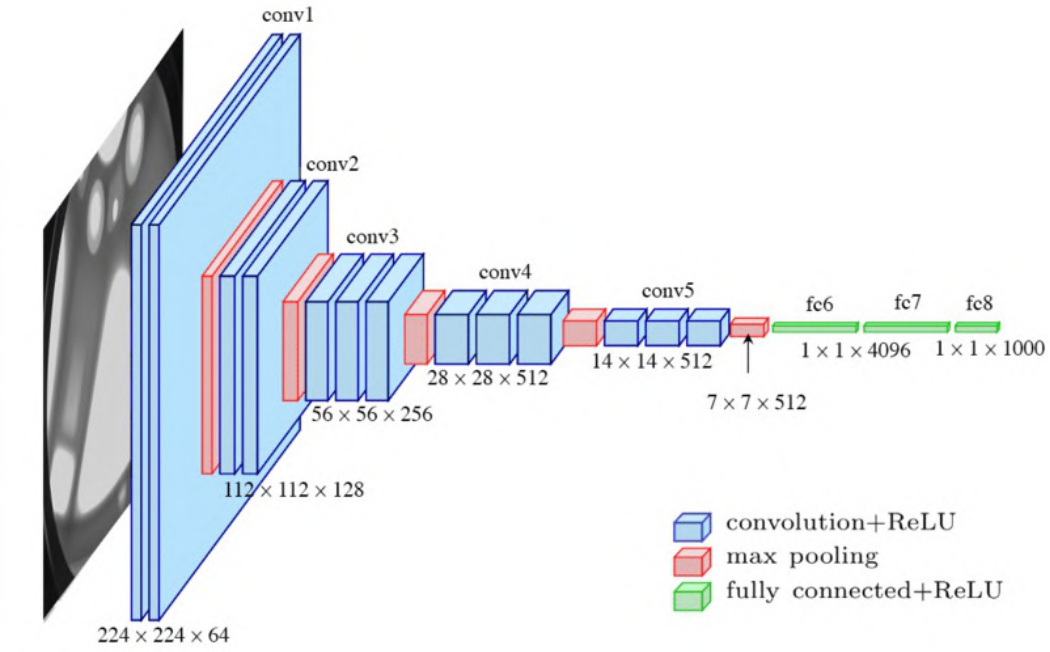


Figure 2.2: The VGG 16 [SZ15] architecture, a conceptually simple yet still widely popular architecture for image classification. Image from Khandelwal[Kha20]

2.1.2 ImageNet and Variants

Deep learning models often contain a large number of parameters and therefore require a large amount of training data. One of the most important datasets for image classification is the ILSVRC 2012 dataset, which contains 1.3 million images labelled into 1000 different categories (or "classes") from ImageNet [DDS+09]. ILSVRC stands for "ImageNet Large Scale Visual Recognition Challenge", which is an annual competition where researchers compete for the highest classification accuracy. In 2012 there was a breakthrough achieved by the CNN *AlexNet* [KSH12], which is commonly attributed to having sparked the

"deep learning revolution" in the following years. ImageNet is organized according to the *WordNet* [Mil95] lexical database. The ILSVRC 2012 dataset has been used extensively in recent years to benchmark new image classification models. Interestingly, it contains a relatively large portion of dogs, 90 classes out of 1000 are different dog breeds. This causes an abundance of feature detectors focusing on dog parts within a trained CNN. Often, the term *ImageNet* gets used in place of the *ILSVRC 2012* dataset, even though strictly speaking the latter is a subset of the former. We will also use this nomenclature here.

The popularity of ImageNet has led researchers to develop variants of the dataset which are aimed at improving or testing the robustness of models. We will look at some of them in Section 3.1. Stylized ImageNet is one particular variant, where the original ImageNet images are replaced with versions generated by *artistic style transfer* [GEB16]. These images thereby contain completely different texture information while preserving the overall shape structure of the original image (as shown in Figure 2.3). Two of the three models whose investigation we facilitate in Perturber have been trained on ImageNet and Stylized ImageNet respectively. The widespread popularity and the significance of ImageNet made it an obvious decision to visualize models trained on this dataset in our visual analytics application.

Figure 2.3: Example from Stylized ImageNet: The image on the left is the original from ImageNet, the other images are variants generated with artistic style transfer. Image taken from [GRM+18].

2.1.3 Transfer Learning

Transfer learning [OBLS14] is a method that is primarily useful to increase generalization capabilites in a few-data setting. A model with a large number of trainable parameters, sometimes hundreds of millions, gets trained on a large, relatively general dataset like ImageNet [DDS+09]. Thereby, it learns a useful feature representation of the input data type, for example for natural images. Then, the model is trained on a smaller dataset of interest, often with early layers not being updated ("frozen"). Freezing early layers keeps them unchanged during subsequent training and therefore prevents the feature extractors trained on the large dataset from overfitting the new, smaller dataset.

The first training procedure is called the *pre-training* step, the second one is called the *fine-tuning* step. As the model has already learned to generate useful features from the data during pre-training, it only needs to learn how to classify the new data from the available features instead of having to learn the features themselves as well. This requires significantly less data, and significantly fewer training iterations.

A typical transfer learning example might be similar to the following setting: The available dataset contains 100 dog- and cat images each, with a resolution of 224×224. The desired architecture is a high-performing CNN like VGG 16 from Figure 2.2. The deep learning engineer might start with a VGG 16 [SZ15] pre-trained on ImageNet and replace the final, 1000 dimensional output layer "fc8" with a 2 dimensional output layer for cat and dog. They would then re-initialize the weights of the fully-connected "fc6" and "fc7" layers, freeze all the convolutional layers, and then train on the available data. The resulting model would learn to use the available features to discriminate cats from dogs by only adapting the weights of the last three layers.

2.1.4 Robustness

Robustness in a broader sense describes the degree to which a model is resistant to image perturbations that do not exist in the training data. These perturbations might be simple ones like color changes (hue, saturation, etc.) and blur, or more complex ones like unusual backgrounds and camera angles. In real-world scenarios, image perturbations might arise, for instance, from faulty white-balance, a de-focused lens, lens flares, etc. [HD18]. Human vision is often not affected by such perturbations. For instance, a human still recognizes a purple cow as a cow, and a cow on a beach might be unusual but would not confuse the human enough to make them think it is something else than a cow. A special type of robustness is adversarial robustness. An adversarially robust model is less vulnerable to adversarial attacks (Section 2.2).

ML practitioners typically try to improve the robustness of a model by data augmentation: Cropping, horizontal mirroring, small rotations, additive noise, and slight color changes are examples of augmentations that are often applied randomly to the training data as part of the input pipeline [SLJ+15]. These augmentations preserve image semantics while modifying pixel values and are therefore an easy way to increase and diversify the amount of training data. Simple data augmentation only provides robustness against the types of modifications performed by the augmenting operations. Researchers work towards developing more sophisticated data augmentation strategies battling robustness issues. For instance, we look at some recent examples of work dealing with more effective data augmentation strategies in Section 3.1.

2.2 Adversarial Attacks

Generally speaking, adversarial attacks are techniques to imperceivably perturb the input data of a predictive model in a way that causes a drastically different result than the

original data. For image classification, adversarial attacks are algorithms to perturb images so that they are perceived completely different to the original image by a model, while being perceptually almost identical for humans. They impose a critical security threat on deep learning models and thus have been investigated extensively in recent years. Notably, researchers have been competing for the strongest and most general attacks and defenses. Adversarial attacks can have two types of objectives. Either they are untargeted, meaning their objective is to suppress the original top prediction, no matter what the new prediction result will be, or they are targeted, aiming to fool the model into predicting a specific class. In our application Perturber, we provide both objectives as an option.

2.2.1 Attack methods

Most adversarial attack methods rely on first-order activation maximization. The gradient of the prediction output with respect to the input image is computed, consisting of values for each input image pixel. The gradient then gets subtracted (untargeted) or added (targeted) to the input, thereby minimizing the original prediction or maximizing the target prediction respectively. Adversarial attacks are usually bounded by a maximum magnitude denoted epsilon (ϵ), which is measured under a specified L_p norm. This epsilon value restricts how much the perturbed image can differ from the original image in pixel space. The most popular norms in this context are the L_2-norm, which restricts the Euclidean distance between the original image and the perturbed image, the L_∞-norm, which restricts the difference independently per-pixel, and the L_0-norm, which counts the number of perturbed pixels.

Some notable attack methods are:

- **Fast gradient sign method (FGSM)** [GSS15]: As the name suggests, FGSM is fast to compute, as it only requires a single iteration. The per-pixel sign of the gradient is multiplied by the attack ϵ, resulting in a maximally perturbed image under L_∞-norm. Therefore this attack is primarily suited for the L_∞-norm bounded setting. This attack can also be applied iteratively to generate a stronger attack.

- **Jacobian-based Saliency Map Attack (JSMA)** [PMJ+16]: This attack iteratively perturbs the pixel with the strongest influence on the prediction. It is therefore suitable as an L_0-norm (number of changed pixels) attack.

- **Projected gradient descent (PGD)**: This method has been shown by Madry et al. [MMS+18] to be the strongest first-order attack method under a certain perturbation budget. It works by iteratively computing the input gradient, adding it to the current input and then projecting the total perturbation back to be of length ϵ. The iterative nature of this method makes it computationally expensive. PGD is highly effective under both L_2- and L_∞-norm.

2.2.2 Defenses

Defenses against adversarial examples can be broadly classified into three categories:

- **Detection methods** try to identify adversarial examples, without correcting the prediction. The adversarial input can then be rejected, preventing harm caused by the attack. Grosse et al. [GMP+17] were able to detect adversarial examples by statistical properties of their activations. Xu et al. [XEQ18] proposed compressing the input image through bit depth reduction or JPEG compression and measuring the difference of the output scores to the original, a techique they call "Feature Squeezing". Many other detection methods have been developed.
- **Gradient masking** methods try to hide the model's gradient from the attacker, to prevent them from computing harmful perturbations. This is done by using non-differentiable operations in the model, by using stochasticity during inference, or by inducing vanishing/exploding gradients through very deep architectures. Gradient masking methods have been shown to be circumventable by Athalye et al. [ACW18].
- **Adversarial training** methods introduce adversarial examples into the training procedure. There are methods that mix natural images and adversarial examples as well as methods that exclusively train with adversarial examples. They are of special interest to us as these methods are similar to the more general concept of robust training with augmented data, like stylized images [GRM+18]. One notable method in this category is PGD-based adversarial training. Madry et al. [MMS+18] provide a proof that is the strongest attack method that relies on gradient information, therefore, training on PGD-generated adversarial examples produces a model that is robust to all first-order attacks.

2.3 Feature Visualization

Feature visualization is a technique that has been developed to provide an insight into what features intermediate or output layers of a neural network have learned. The first convolutional layer has three input channels and can thus be directly visualized simply by showing the weight kernel as an RGB image, as shown in Figure 2.4. This is because in 2d convolution, each feature's kernel is 3 dimensional with a shape of *kernel-width* $\times$ *kernel-height* $\times$ 3, just like an RGB image. Later layers require a more involved visualization generation process.

A simple approach is activation maximization [EBCV09], which is closely related to adversarial attacks. Activation maximization iteratively optimizes the input image by gradient ascent. The optimization objective can be maximizing the activation of either the desired neuron, a set of neurons like a single featuremap ("channel"), or all featuremaps of a layer ("deep dream") [OMS17], among others. Perturber makes use of feature visualizations that maximize a single neuron's activation.

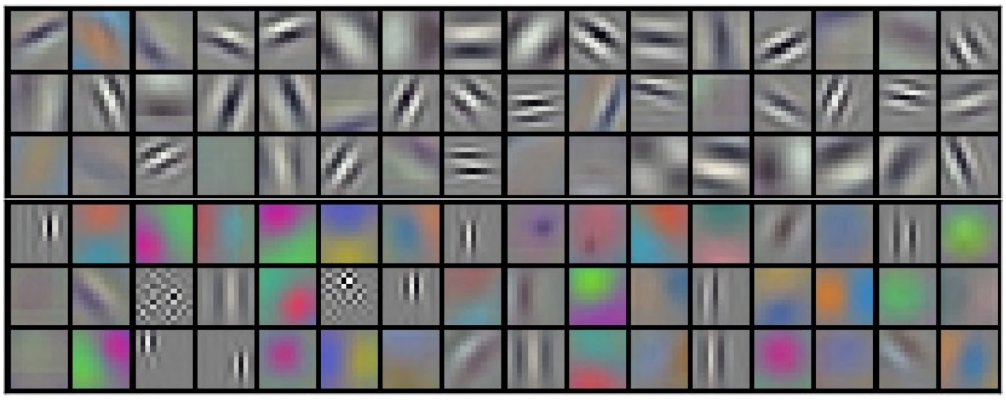
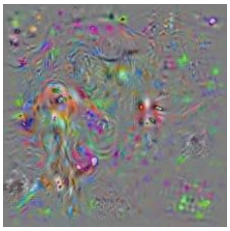
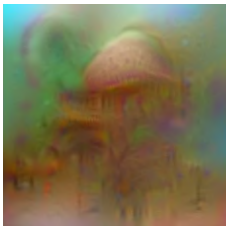


Figure 2.4: 96 filter kernels learned by the first convolutional layer of AlexNet. The kernels have shape $11 \times 11 \times 3$ [KSH12]. Image from [KSH12].



(a) Class "Golden Retriever" visualized by Nguyen et al., generated using gradient blurring regularization [NYC15].



(b) Latent code image reconstruction by Mahendran and Vedaldi with total variation encouraging piece-wise constant patches [MV15].

(c) Class "Bramble" visualized by Nguyen et al. by optimizing the latent code of a deep generative network [NDY+16].

(d) Class "Redshank" visualized by Nguyen et al. by optimizing the latent code of a denoising autoencoder [NCB+17].

Figure 2.5: Four examples of regularization and parameterization techniques for feature visualization.

Naive activation maximization usually leads to a noisy image, similar to an adversarial perturbation vector and thus is not very useful to humans for understanding the concept of the respective feature. With "feature", we mean the set of patterns that cause strong activation at the respective neuron, for instance "horizontal lines" or "left oriented dog heads". A wide range of regularization- and parameterization techniques have been developed to mitigate this issue by enforcing properties of natural images. We discuss some examples in the following.

2.3.1 Regularization and Parameterization

Regularization and parameterization techniques for feature visualization range from explicit penalization of high frequencies by total variation [MV15] to feeding the input

Figure 2.6: Feature visualizations (top) and dataset examples (bottom) for Neuron 6 of Layer "mixed4e" from Inception V1. The feature visualizations have been generated from different random initializations. Image consists of screenshots from [Ope].

image through a generative model. The former encourages images to consist of piecewise constant patches. The latter constrains the generated visualizations to the output manifold of an auto-encoder or of a generative adversarial network (GAN) [GPAM$^+$14] that have previously been trained to generate natural images [NDY$^+$16].

Another "extreme" form of regularization are dataset examples, which implicitly are constrained to natural images from the training set. Dataset examples are generated by recording the activation values of image patches from a dataset for the respective neuron, for instance the ImageNet validation set. The image patches which activate the neuron the most are then used for visualization purposes. They have been shown to be particularly well suited for conveying concepts to humans [BZS$^+$20], but fail at revealing feature properties that deviate from those found in natural images. Figure 2.6 shows a comparison of feature visualizations and dataset examples for the same neuron.

In our work we follow the techniques used in the Circuits project [CCG$^+$20] and OpenAI microscope [Ope]. They use transformation robustness as a regularizer and a color- and frequency decorrelated image parameterization. Transformation robustness stochastically applies small transformations to the input in each optimization step to find an image that achieves the desired activation, even when slightly transformed. This helps smoothing out checkerboard artifacts and high frequency noise [OMS17]. Color decorrelation is achieved by computing the correlation among the RGB values over the whole dataset

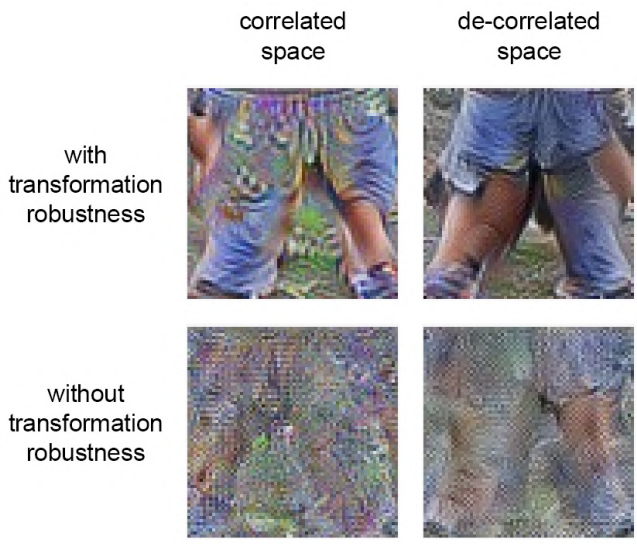


Figure 2.7: Feature visualizations of Inception V1 neurons with multiple regularization and parameterization configurations, assembled from [OMS17].

Figure 2.8: Feature visualizations from various layers of Inception V1, starting on the left with low-level feature detectors with small receptive fields from early layers, to high-level detectors with large receptive fields from later layers on the right.

and multiplying the input image by the square root of the color correlation matrix before feeding it into the CNN [OMS17]. Spatial decorrelation is done by parameterization of the image in Fourier space and applying an inverse fast Fourier transform (iFFT) before feeding the image to the model. This helps the gradient to be more evenly distributed among spatial image frequencies, thus achieving a more natural appearance. Figure 2.7 shows a juxtaposition of feature visualizations generated with and without Fourier parameterization and transformation robustness.

Examples of feature visualizations from various layers can be seen in Figure 2.8.

2.3.2 Polysemantic feature detectors

While feature visualization can generate an image that maximizes selected activations, many neurons play multiple roles at once and respond strongly to a large variety of

patterns. This might be simply an invariance to transformations, like Neuron 55 from Layer "mixed4b" of Inception V1, which responds to both left-arching and right-arching curves (Figure 2.9b). There can also be more peculiar combinations like found in Layer "mixed4e", Neuron 55, which responds strongly to cats as well as to cars (Figure 2.9a). Visualizing this polysemanticity can be achieved by optimizing multiple randomly initialized input images concurrently and introducing a diversity term into the optimization objective. A diversity term can be any term that encourages the individual feature visualizations to be dissimilar. In Olah et al. 2017 [OMS17], this term is computed by calculating the negative accumulated pairwise cosine similarity of the gram matrices of each generated image. The gram matrix G is calculated by

$$G_{i,j} = \sum_{x,y} l[x, y, i] \times l[x, y, j] \tag{2.1}$$

where x and y are the spatial locations of layer l and i, j are channels. The gram matrix thus contains the pairwise dot products for each flattened featuremap of the respective layer output. It has been used in style transfer [GEB16] to capture the "style" of an image, independent of spatially defined content. Maximizing the style difference between generated feature visualization by minimizing the negative cosine "style-similarity" leads to a diversified batch of feature visualizations. Using the "style" as a comparison metric here instead of a direct pixel loss (like pixel-wise L1) is very important, as illustrated by the following example: An image with alternating black and white vertical lines of one pixel width is completely dissimilar to the same image shifted by one pixel to the right, under pixel-wise L1 distance, even though the styles of the two are virtually identical. Besides diversified feature visualizations, dataset examples can also be an effective way to look at polysemantic neurons. Polysemanticity adds an additional layer of visual complexity, thus in our visual analytics application, we refrain from depicting multi-image visualizations. This comes at the cost of slightly less meaningful visualizations for deeper layers.

(a) Four feature visualizations of Neuron 55 of Layer "mixed4e" from Inception V1. The neuron responds to cat- or fox-like animals, but also to cars (left).

(b) Four feature visualizations of Neuron 226 of Layer "mixed4b" from Inception V1. The neuron responds both to left- and right-arching curves.

Figure 2.9: Feature visualizations with "diversity" term.

CHAPTER 3

Related Work

In this chapter we describe the state-of-the-art of related research fields, namely robustness and adversarial examples, as well as visual analytics in deep learning. We start by looking at recent work in the field of robustness, then we show some examples of work about visual analytics in deep learning, a rather new research direction with many possibilities yet to be explored.

3.1 Robustness in Deep Learning

Recent work on robustness has revealed the highly brittle predictive performance of convolutional neural networks. Interestingly, while a large portion of research has focused on studying the existence of adversarial examples, it appears that investigating the more general concept of robustness has not sparked as much interest among deep learning researchers. Perturber has been designed to facilitate the visual investigation of CNN behaviour both under adversarial attacks and under more general perturbations.

The strong interest of the deep learning community in adversarial examples has presumably been sparked in 2014 by Szegedy et al. [SZS+14], who first demonstrated the existence of adversarial examples. Apart from showing that CNNs can easily be fooled by humanly imperceptible perturbations, they also demonstrated their generalizability to other network architectures trained with the same dataset. This means that an adversarial attack generated from network A also fools network B if they are both trained on the same dataset. Subsequently, researchers have been trying to find defenses against adversarial attacks. For example, defensive distillation [PMW+16] trains a second neural network on the logits of an initial trained classifier as an additional barrier for an adversary. The authors argue that the gradient of the class probabilities with respect to the input image is much lower in the distilled network, making a gradient based attack significantly harder. Feature squeezing [XEQ18] compares the activations of the original image to one that has been degraded by bit-depth reduction or JPEG compression to detect adversarial

Figure 3.1: (a) Shows an example from the *backgrounds challenge* (Image from the authors' blogpost [EIMX20]). (b) Shows a cat with an elephant texture, which was used by Geirhos et al. in their experiments comparing human- and CNN sensitivity to texture versus shape. Image from Gheiros et al. 2019 [GRM+18].

examples. Various researchers have also proposed training on adversarial examples to improve robustness to adversarial attacks. Goodfellow et al. [GSS15] and Madry et al. [MMS+18] present examples of such work. Most of these defenses have been proven insecure [CW17]. To our knowledge, the first method confirmed by a third party to be robust against adversarial attacks is PGD adversarial training [MMS+18]. Here, the model is continuously trained on adversarial examples generated by PGD, which has been theoretically shown to be the strongest gradient-based attack method.

Although "robustness" has almost become synonymous with "adversarial robustness", some researchers have called for a shift of robustness research to the more general concept of robustness [GH19]. They argue that with adversarial examples being only the most extreme cases where the brittleness of CNNs is exposed, a large amount of other perturbations exist that equally fool CNNs while having negligible effect on human vision. These perturbations can include camera defects, color shifts, or the addition of deterring objects to the scene [RZT18]. Xiao et al. [XEIM20] have found that objects in front of unusual backgrounds are often misclassified, and created the *backgrounds challenge* for researchers to benchmark their models in this regard. Geirhos et al. [GRM+18] have changed the texture of images while preserving the shape via style transfer and thereby revealed a strong difference between human vision and CNNs regarding focus on texture versus shape.

Others have investigated CNNs' tendency to learn surface statistical properties [JB17], or compared differently trained CNNs regarding their sensitivity to perturbations of varying frequencies. They discovered that adversarially trained CNNs are highly sensitive to low frequency perturbations (for example "fog"), whereas standard trained CNNs are more sensitive in the high frequency spectrum [YLS+19].

To increase general robustness of CNNs, sophisticated data augmentation techniques

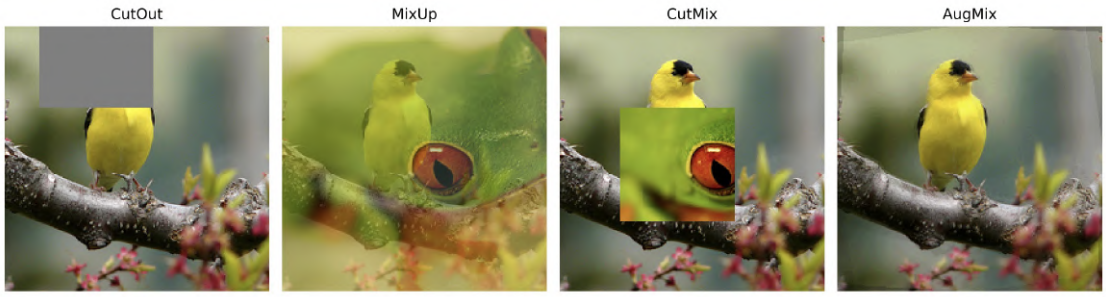


Figure 3.2: Examples of recent data augmentation strategies that go beyond simple geometric or color transformations. Figure taken from Hendrycks et al. 2020 [HMC+20].

have been developed. CutOut [DT17] randomly masks part of the image input during training and has been shown to increase robustness of CNNs. CutMix [YHO+19] is an augmentation where patches of another training example are pasted into the current image, and the labels are mixed proportionally to the area of the patches. The authors demonstrate an increased model robustness against various input corruptions when using this technique. MixUp [ZCDLP18] trains a neural network on convex combinations of pairs of examples and their labels and thereby regularizes its behavior towards more linearity in-between training examples. AugMix [HMC+20] uses a diverse set of augmentations stochastically in combination with a Jensen-Shannon Divergence consistency loss, while also mixing multiple augmented images and thereby achieves state-of-the-art performance. AutoAugment [CZM+19] is a procedure to automatically search for better data augmentation policies. The authors achieve state-of-the-art accuracy on multiple datasets, including ImageNet, by using their technique. They also show that their policy learned on ImageNet transfers well to other large-scale image datasets.

To benchmark ImageNet-trained models' robustness, various ImageNet adaptations have been designed. *ImageNet-C* [HD18] applies 75 common visual corruptions to standard ImageNet. These corruptions include various types of noise, blur, color transformations and spatial deformations as well as pixelation and lossy compression. Some examples can be seen in Figure 3.3a. *ImageNet-P* [HD18] introduces fine-grained progressions of 10 visual corruptions. These include translation, affine and perspective transformations, as well as additive effects like spatter dripping down the lens. These progressions can be used to test the invariance of a model to small continuous changes, similar to what might occur in a real video stream. *ImageNet-A* and *ImageNet-O* [HZB+21] contain "natural adversarial examples", which are natural images that are classified incorrectly with high confidence. These can be for instance a squirrel image that gets confused with a sea lion or a dragonfly image that gets mistaken for a manhole cover because of its grid-like background, like seen in Figure 3.3b. While ImageNet-A contains images from the ILSVRC2012 classes, ImageNet-O contains out-of-distribution images whose classes are not found in the training dataset. Examples from ImageNet-O can be seen in Figure 3.3b. *ImageNet-R* [HBM+21] contains human-created renditions of ImageNet samples,

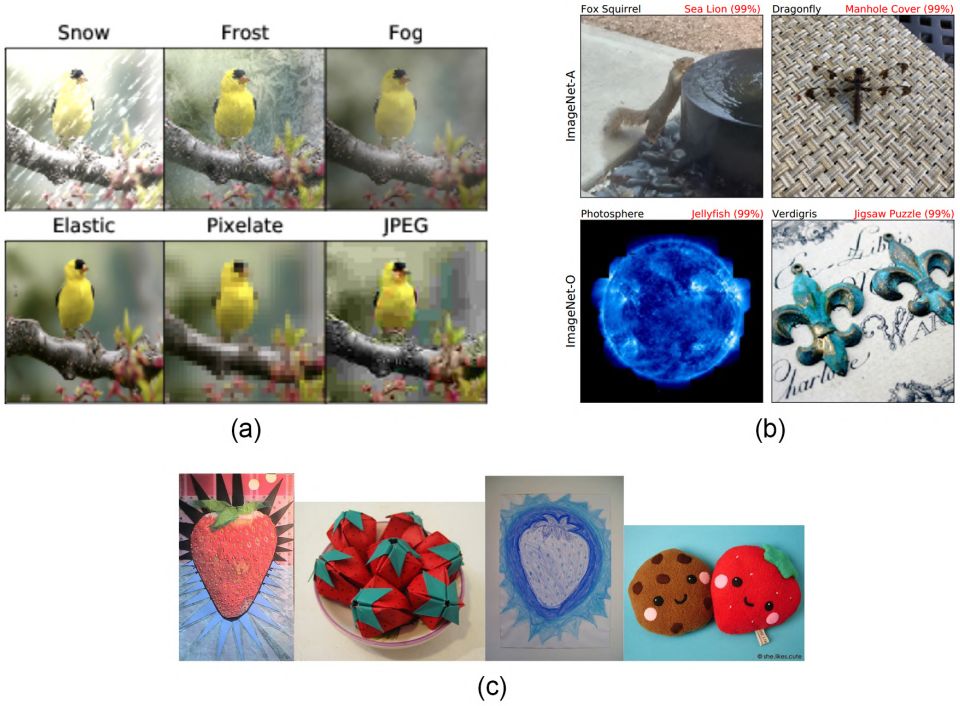


Figure 3.3: (a) Example corruptions from ImageNet-C. (b) Example "natural adversarial images" from ImageNet-A and ImageNet-O. (c) Example images from ImageNet-R.

for example drawings, paintings, origami, embroidery, sculptures, etc. (Figure 3.3c). These ImageNet adaptations offer a way to test a network against visual perturbations, albeit with a more systematic and less interactive approach than our work.

3.2 Visual Analytics in Deep Learning

In recent years, a vast amount of work on deep learning visualization has been published. We refer the reader to the recent survey by Hohman et al. [HKPC19] for a comprehensive report on deep learning visual analytics. The relevant visual analytics work presented in the following paragraphs can be roughly grouped by the target group, divided into experts and non-experts, as well as the task they have been designed for: Education, model understanding and model improvement.

Visual analytics applications designed for non-experts usually require minimal prior deep learning knowledge and can be used for educational purposes. They are often designed as "playgrounds", where a user can interact with a simple GUI without having to write any code. Furthermore, many of these applications include educational content, like text explanations, directly into the interface. Notable examples are *TensorFlow Playground* [SCS+17] and *GANLab* [KTC+19]. TensorFlow Playground supports the

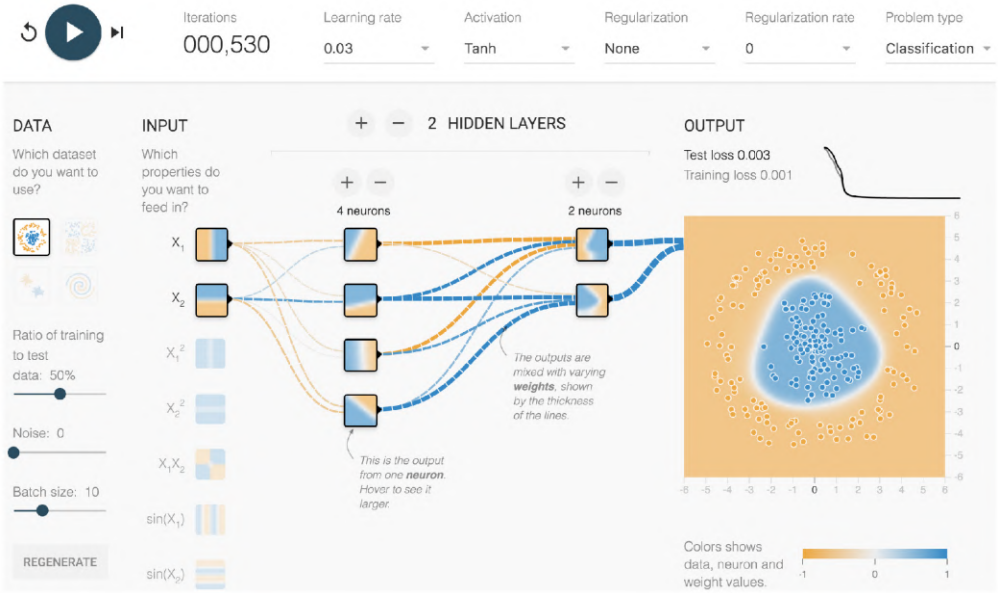


Figure 3.4: Interface of TensorFlow Playground. Image from [SCS+17].

interactive modification and training of deep neural networks in the browser. A multi layer perceptron's (MLP) architecture can be defined via controlling the number of layers, number of units per layer, and the activation funtions. *GANLab* similarly supports interactive experimentation with Generative Adversarial Networks and allows users to slowly step through the training iterations of generator and discriminator. Both applications are accompanied by multi-paragraph explanatory text. *CNN Explainer* [WTS+20b] visually explains the inner workings of a CNN. It depicts connections between layers and activation maps and allows the user to choose the input from a pre-determined set of images. Like these examples, Perturber is also an application that runs in the web browser, but allows users to manipulate input images fully interactively and to observe model responses simultaneously.

Harley [Har15] published an online tool where users can draw digits interactively. The drawing is fed into a small MNIST-trained network and the responses of all neurons are visualized in real-time. This is very similar to our work in that the application allows the user to manipulate the input image while observing the network output. The application provides an MLP, as well as a CNN. *Adversarial Playground* [NQ17] enables users to compute adversarial attacks and instantly observe the changing predictions of a *simplistic* MNIST-trained network. Contrastingly, Perturbers purpose is to facilitate the user's understanding which perturbations have a large impact on a *complex* CNN, and why. Therefore, we work with an ImageNet-trained Inception V1, complex 3d input scenes including animals or man-made objects placed in various environments, and a large collection of perturbation methods to manipulate the input scene and to thereby attack the model.

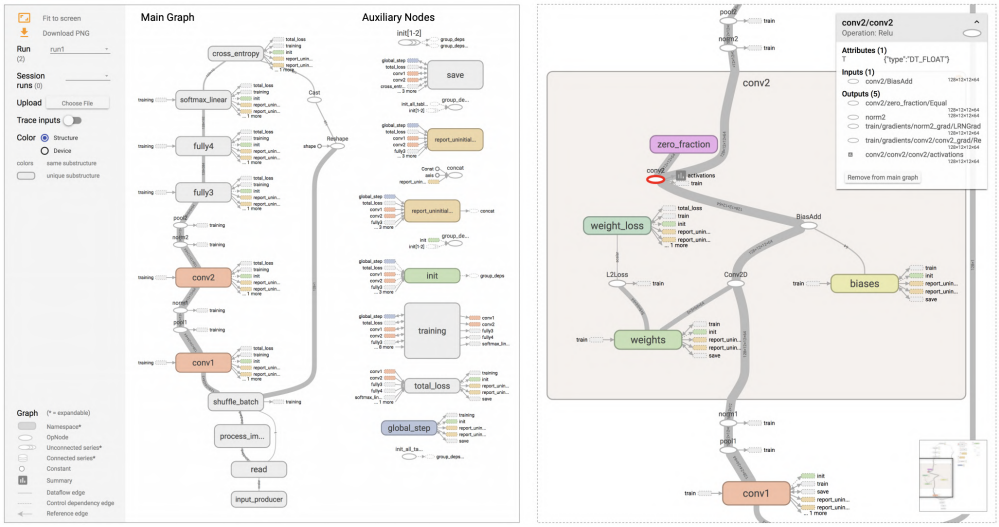


Figure 3.5: Interface of the TensorFlow Graph Visualizer. On the left an overview of the whole graph structure is shown, on the right a subscope has been opened to show the contained sub-graph in more detail. Image from [WSW+18].

Visual analytics tools for experts have been created with a wide range of purposes. Graph structure visualizations, like the *TensorFlow Graph Visualizer* [WSW+18], provide users with a depiction of their model's structure, which is often a highly complex network of layers and connections. Others focus on training and record detailed metrics over the course of the training process. *DeepEyes* [PHVG+18] and *DeepTracker* [LCJ+19] are examples of applications that fall into this category. DeepEyes focuses on "debugging" problems such as degenerated filters or redundant layers whereas DeepTracker can help detecting anomalous training iterations. *ExplAIner* [SSSEA20] is implemented as a Tensorboard plugin and helps users to better understand their models by providing a collection of various existing explainability methods.

While the before described tools all facilitate the examination of a single model, others support the comparison of multiple models. These can have a shared architecture with different weights, or different architectures. For example, *REMAP* [CPCS20] allows users to efficiently improve model architectures. This is facilitated by ablation (i.e. removing single layers of an existing model) and variation (i.e. creating new models by replacing layers). The users can then compare the found model architectures regarding their performance through various metrics. Ma et al. [MFH+20] designed multiple coordinated views letting experts analyze model behaviors after fine-tuning. Users can visually compare the model between original and fine-tuned state through a matrix of neuron similarities. Their application also lets users inspect extracted features through feature visualizations. *CNNComparator* [ZHP+17] focuses on comparing the architecture and the output of a selected input image between two CNNs. In contrast, the focus of our work lies on interactive modification of the input image and instantaneous comparative inspection

of the network responses. Rather than letting the user select pre-determined input images with a ground-truth class label from a fixed set, we therefore let users *generate and perturb* input images from 3d scenes in a playground-like manner. Additionally, users can observe the real-time classification predictions and real-time activation maps of individual, relevant neurons.

Prospector [KPN16] has been designed for tabular data and can show the influence of a feature on the prediction while keeping the other features fixed, a visualization known as *partial dependence plot*. It also allows interactively changing features of individual datapoints through a slider-based interface, where each slider manipulates one feature. The *what-if-tool* [WPB+20] similarly allows modifying feature values while observing the changing prediction, but also supports the user in assessing fairness as well as balance with respect to feature distributions in the dataset. *NLIZE* [LLL+18] specializes on natural language processing and, in addition to supporting interactive input perturbations, allows the user to interactively modify the intermediate results within the model. *LIME* [RSG16] is a generally applicable framework, where perturbed samples around an input point of interest are used to facilitate local interpretability. LIME takes advantage of the fact that while ML models usually have a highly non-linear prediction landscape, the neighbourhood around a single datapoint is often linear. This allows to approximate the original model within this neighbourhood by a linear model, which is highly interpretable by default. LIME is purely input/output based and thus is suitable for explaining black-box models. Neither of these systems lets the user interactively probe an image classification model by perturbing an input image like Perturber. While LIME supports investigating image classifiers, the computation is performed offline and the result is then presented to the user afterwards.

To help explaining a CNN, there are sophisticated techniques to visualize the learned features of the model's hidden neurons. *Feature visualization* is such a method and is based on activation maximization [EBCV09] in combination with a selection of regularization techniques [YCFL15, OMS17].

Feature visualizations have been used as the base technique for a multitude of visual interfaces. For instance, Carter et al. [CAS+19] used feature visualizations to depict the feature space of layers of interest by projecting sampled activations onto the 2d plane through dimensionality reduction (Figure 3.6). Cammarata et al. [CCG+20] used feature visualizations to find and investigate causal relationships between neurons in CNNs. OpenAI Microscope [Ope] is a system that comprehensively documents the response patterns of individual neurons in multiple large CNNs.

In visual analytics tools, feature visualizations have been used to compare learned features before and after transfer learning [MFH+20] or to visualize a graph of the neurons with high impact for a selected target class and their connections [HPRPC20]. Similarly, *Bluff* [WTS+20a] depicts a graph where neurons are represented by their feature visualizations to visualize neurons which are most impacted by adversarial attacks. In contrast to Perturber, the adversarial attacks visualized by Bluff are precomputed. Its interface is shown in Figure 3.8.

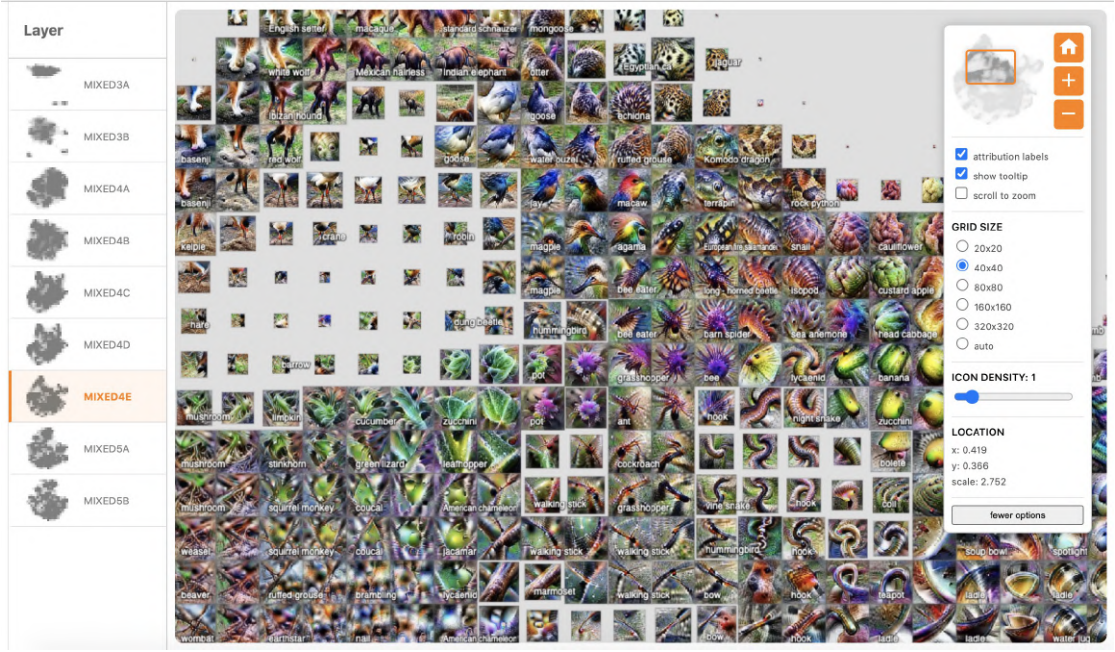


Figure 3.6: Activation atlas of Layer "mixed4e" from Inception V1. Screenshot from Carter et al. 2019 [CAS+19].

Saliency maps (or *attribution maps*) show the significance of the input image's regions with respect to the selected target class or network component [SVZ14] and have also become popular interpretability tools. Saliency maps and other gradient-based methods like *LRP* [LBM+16], *Integrated Gradients* [STY17], *SmoothGrad* [STK+17], or *Grad-CAM* [SCD+17], however, necessitate a back-propagation pass which comes with significant computational cost. A computationally less costly technique is to directly visualize the forward-propagation activations of selected feature maps in intermediate layers. For example, the *DeepVis Toolbox* [YCFL15] shows live visualizations of CNN activations from a webcam feed. In contrast to attribution methods, this solution visualizes which image regions activate individual neurons but provides no information about the significance to the predicted class. The goal is to get a general intuition what features a CNN has learned.

AEVis [LLS+18] shows "datapath visualizations", allowing users to follow the effects of adversarial attacks on significant neuron's activations through the hidden layers of a CNN. Datapaths in AEVis consist of critical neurons that are responsible for the corrupted predictions. Computing the subset of critical neurons, which minimizes the prediction change when isolated from the rest of the network, is one of the main contributions of AEVis, setting its datapaths apart from the simpler pathways in Bluff. While *AEVis* [LLS+18] is restricted to a fixed set of example input images for interactive research, the *DeepVis Toolbox* [YCFL15] can process live webcam stream and visualizes the hidden network layers responding to a continuously changing input. Similarly,

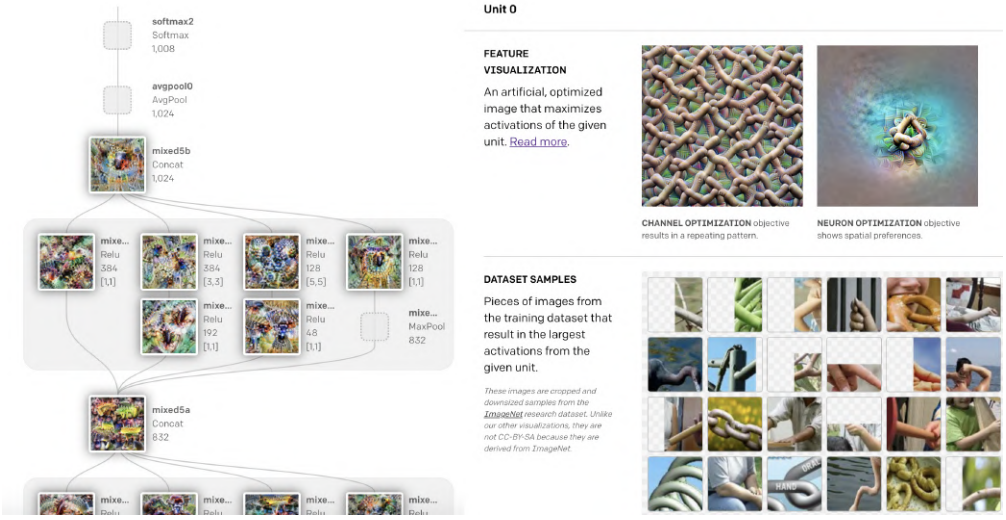


Figure 3.7: Screenshot of the OpenAI Microscope. The left side shows a conceptual depiction of the model architecture with multiple parallel layers. Each layer is represented by a single feature visualization. When the user clicks on a layer node, they get presented with a grid of feature visualizations for each neuron of the selected layer (not shown). When they chose a neuron, they get shown a detail view (right side), which shows a larger image of the feature visualization with both "channel" and "neuron" objective, dataset examples, and tuning curves for various input variations (not visible in the screenshot). Image is a screenshot of [Ope].

Perturber shows activations and predictions based on a live input. In contrast to the *DeepVis Toolbox* however, Perturber renders input images from a steerable 3d scene and provides a rich collection of input perturbation tools. Feature visualizations are directly juxtaposed to activation maps, improving their readability. Additionally, Perturber facilitates direct comparison between standard and robust models.

3DB [LSI+21] is a recently published framework for systematically analyzing and debugging vision models using photorealistic rendering. The framework is based on the same basic idea as Perturber: Varying a rendered 3d scene to analyze the responses of a computer vision model to various perturbation parameters. Figure 3.10 shows an overview of the framework. Compared to 3DB, Perturber trades off flexibility and systematic control for interactivity. We hypothesize that an interactive tool like Perturber could be highly effective for generating hypotheses to then be systematically investigated with 3DB.

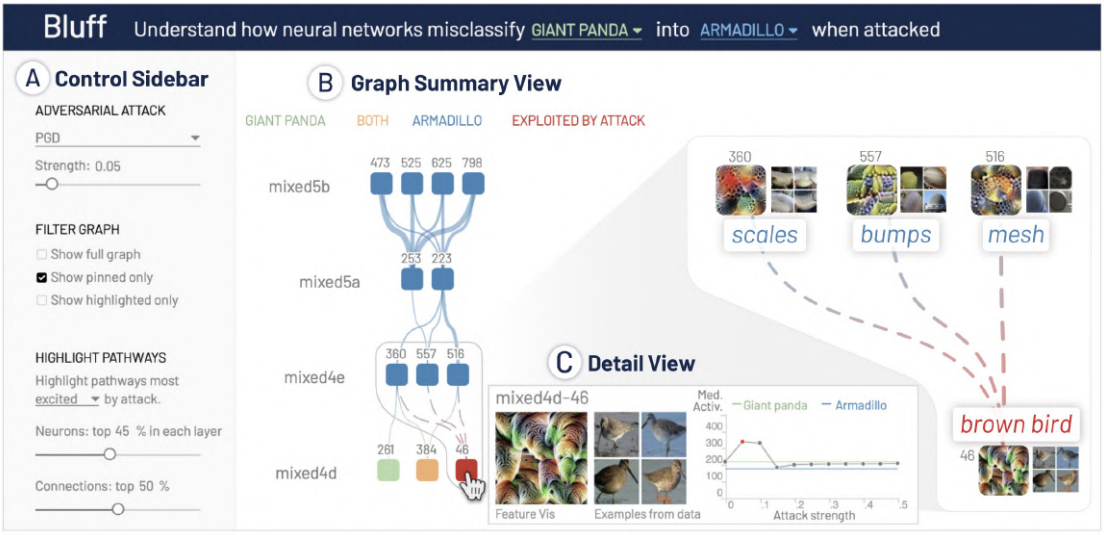


Figure 3.8: Bluff user interface. (A) With the Control Sidebar, the user can select which data to include and highlight. (B) The Graph Summary View depicts a graph where pathways through neurons that were most activated or changed during an attack are highlighted. (C) The Detail View displays feature visualizations, dataset examples, and activation patterns over attack strengths for a hovered-over neuron. Figure from Das et al. 2020 [WTS+20a].

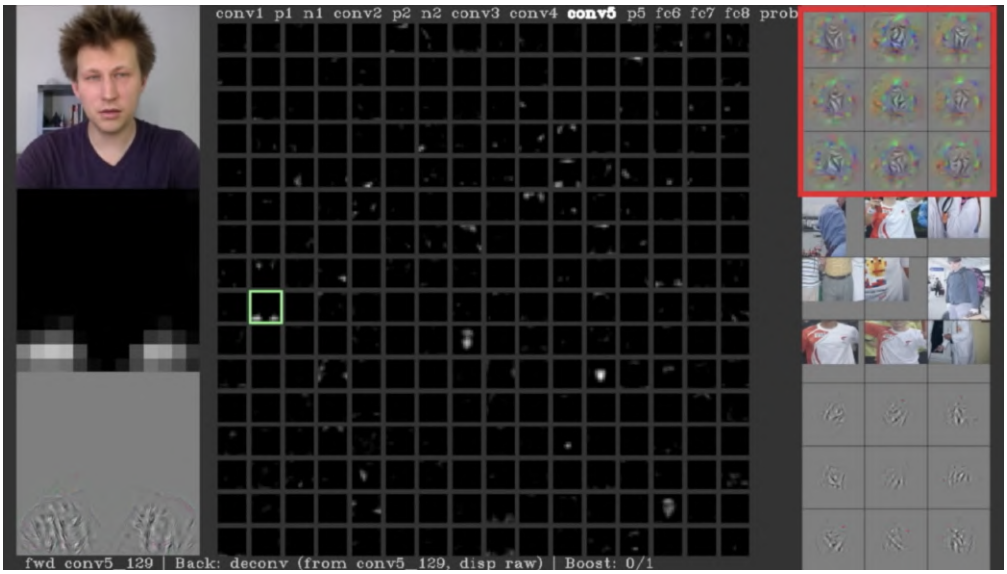


Figure 3.9: Screenshot of the deep visualization toolbox [YCFL15]. In the top-left, the current image of the webcam-stream is shown. In the large center area, the activation maps of the selected convolutional layer ("conv5") are shown. On the right side, various visualizations depicting what the selected neuron responds to are shown.

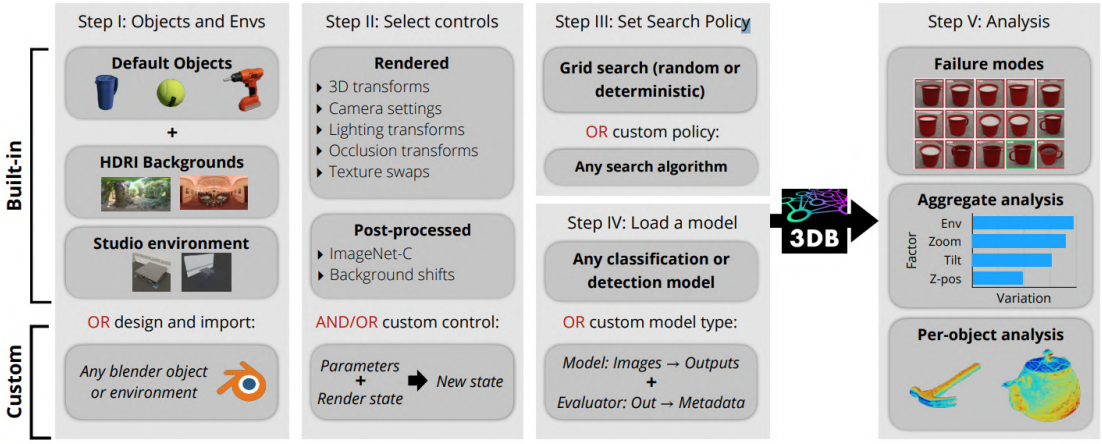


Figure 3.10: Schematic overview of the 3DB framework. In a first step, the 3d objects and environments are specified. Then, controls for perturbing the scenes are selected (Step II), together with a search policy over these controls (Step III), and a computer vision model to be investigated (Step IV). Based on the search policy, the framework selects the scenes and the settings to render. Finally, the computer vision model is evaluated on the rendered images and an analysis consisting of various visualizations and metrics is presented (Step V). Figure from [LSI+21].

CHAPTER 4

Preliminary Analysis

Our visualization tool in its final form is the result of an iterative research process. We started on the premise that feature visualization can give new insights about the differences between standard- and adversarial training (H1), but a lot of intermediate experiments were necessary to arrive at the final design. The results of these experiments had significant influence on the subsequent design decisions.

Our early experiments mainly involved training CNNs with different configurations and then generating feature visualizations for checkpoints throughout the entire training process to compare the different training procedures visually. We summarize the results from these experiments in Section 4.1.

In order to compare feature visualizations between different training methods effectively, we needed to align their feature representations, pushing them to learn similar feature detectors at the same neuron locations. It turns out that transfer learning is effective at preserving feature representation when training with a variant of the dataset used for initial training. Section 4.2 contains more details about transfer learning and its behaviour regarding feature preservation.

As there has been an ongoing effort to investigate and categorize the role of single neurons by Cammarata et al. [CCG+20], we adopt their model of choice, the original Inception V1 network [SLJ+15] instead of ResNet [HZRS16], which we used for previous experiments. This enables us to integrate their already identified neurons and neuron categories into our visualization tool.

The increased dependence on shape and low frequency features versus texture is a characteristic that networks trained with Stylized ImageNet by Geirhos et al. [GRM+18] share with adversarially trained networks. In their experiments, they showed that training on their heavily augmented version of ImageNet leads to models that are more dependent on shape and less dependent on texture, which is in contrast to standard-trained models. In Section 4.4, we discuss our experiments with training on Stylized ImageNet, and take

a look at the feature visualizations we generated for those models. One might assume that they are similarly rich in low frequency structure like those from an adversarially trained model, but this is not the case.

4.1 Initial Experiments

In this section we present our experiments aimed at analyzing how feature detectors in CNNs develop during training. To be able to make a useful comparison between different training configurations, we perform each run with identical initialization and identical data curriculum, preserving the order of the training examples shown to the network. Early tests showed that even with the exact same initialization, a different order of training images leads to the divergence of weights after a single-digit number of iterations. In the presented experiments, if not stated otherwise, we train a *ResNet 18* network on *Restricted ImageNet* [TSE+19], a subset of ImageNet grouped into nine animal super-classes ("Dog", "Cat", "Frog", "Turtle", "Bird", "Primate", "Fish", "Crab", 'Ìnsect'). For weight initialization of the convolutional layers, we use a variance scaling initializer with weights normalized by the number of output nodes ("fan out" mode).

With our initial experiments we look at the following questions regarding feature development during CNN training in general and during adversarial training as a special case:

- **When** during training do feature detectors diverge in standard vs. adversarial training?
- **How** do feature detectors develop during training in general, when viewed through the lens of feature visualization?
- **What** is the influence of the perturbation epsilon (explained in Section 2.2.1)? Are there any feature visualization characteristics that correlate with the perturbation epsilon?

4.1.1 When Does Adversarial Training Diverge From Standard Training?

To answer this question, we look at the weights of the first convolutional layer. In ResNet 18, this layer has 64 3d-kernels of shape $7 \times 7 \times 3$. This means that we can visualize the weights directly as 64 RGB images with a spatial resolution of 7×7, giving us an undistorted view on how the weights of the first layer develop.

Figure 4.1 shows the development of six of the 64 kernels throughout standard training (epsilon 0) and adversarial training with epsilons 0.001, 0.002, 0.05, 0.1 and 0.33 under L_2 norm. Recall that the epsilon value denotes the magnitude of the attack perturbation, which is the difference between the attacked image and the original. We can observe that in most neurons, a difference is clearly visible within the first 10 training steps. In

the orange-marked kernels, there is a relatively large non-linear difference between them. In the red ones, the difference is slightly lower at iteration 10, but nonetheless becomes striking later during training. In the cyan-marked kernels, the very low epsilon values both lead to low-magnitude kernels. Looking at the final kernels (green boxes) after 30K training steps and comparing them to the kernels produced with different epsilon values does not reveal any intuitive relation between adversarial training epsilon and convolution kernel. It is not surprising that a non-linear optimization process, like training a CNN, leads to chaotic results over 30K steps. Nevertheless, the early divergence within *the first few training steps* hints at the strong influence of adversarial perturbations on the training process.

4.1.2 Development of Features During Training

To investigate feature development during training, we generated feature visualizations for an arbitrary subset of neurons from multiple layers of a *ResNet 50* model. We trained the model with standard training and adversarial training. We did not fully train the adversarial model as this would be prohibitively expensive computationally. Instead we trained the standard model until the training error curve flattened significantly, then trained another model adversarially for a similar amount of training steps. We made all of our experiments on adversarial training with L_2-bounded perturbations. In this particular experiment, we used an epsilon value of 1.0. We compare the development of the feature visualizations in Figure 4.2a and Figure 4.2b. The training step numbers do not match exactly, as we saved checkpoints during training based on time intervals, not on training step number. Although not ideal, this does not perturb the visual result significantly enough to require a full repetition of the experiment.

We generated the feature visualizations without Fourier parameterization to highlight the phenomenon of adversarially-trained models having gradients with more low-frequency structure, as described by Tsipras et al. [TSE+19]. This phenomenon is the most striking difference between the two visualizations and becomes apparent in the figures starting with the third row (∼1K training steps). We can also observe the effective receptive field growing during training in intermediate and late layer neurons, an effect that has been theoretically described by Luo et al. [LLUZ16].

Although this comparison of feature visualizations during training confirmed known phenomena, we were not able to deduce any hypotheses regarding the difference between standard- and adversarial training. The fact that, in spite of identical initialization and training schedule, the features diverge very quickly (as seen when comparing Figure 4.2a and Figure 4.2b), led us to explore ways to better align the feature representation for a direct comparison. We explain our findings in this direction further in Section 4.2.

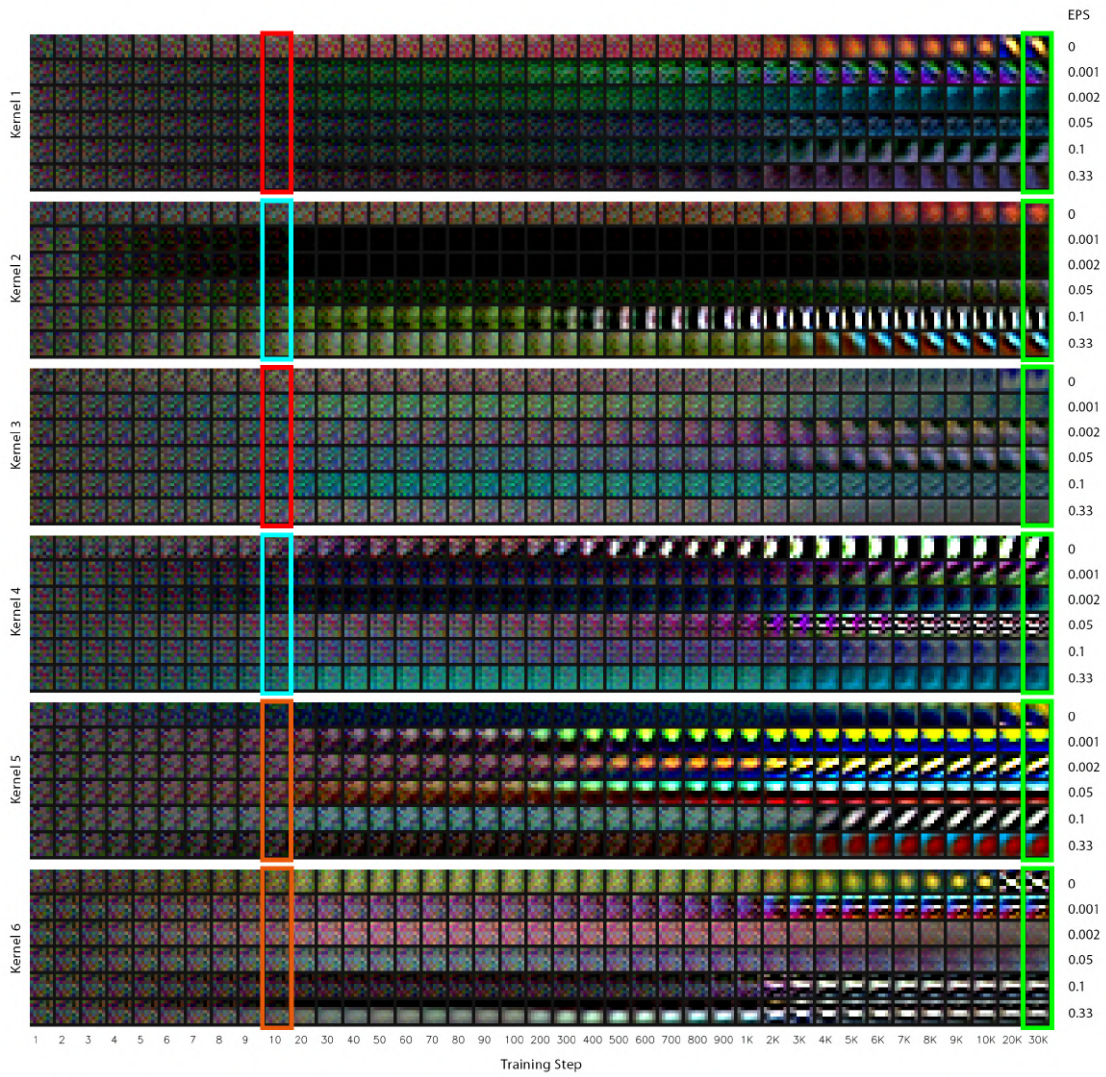
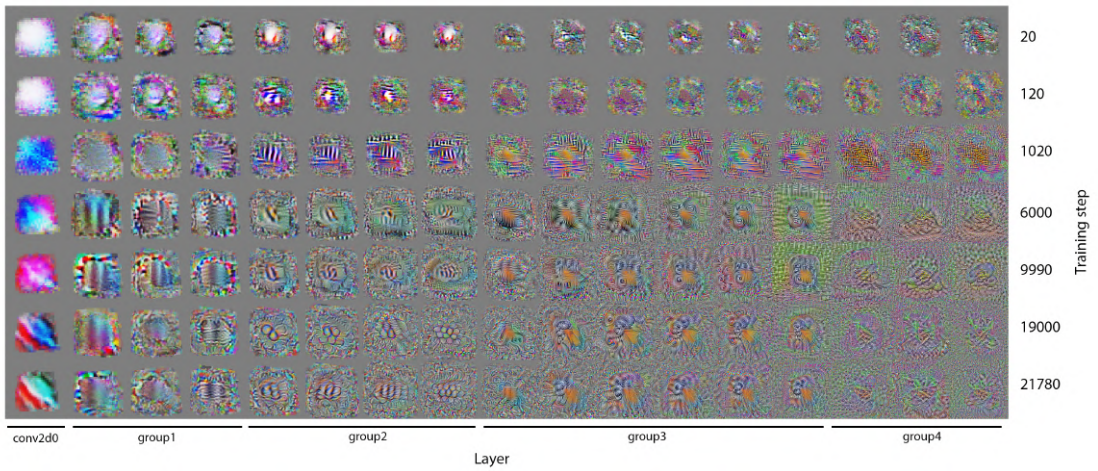
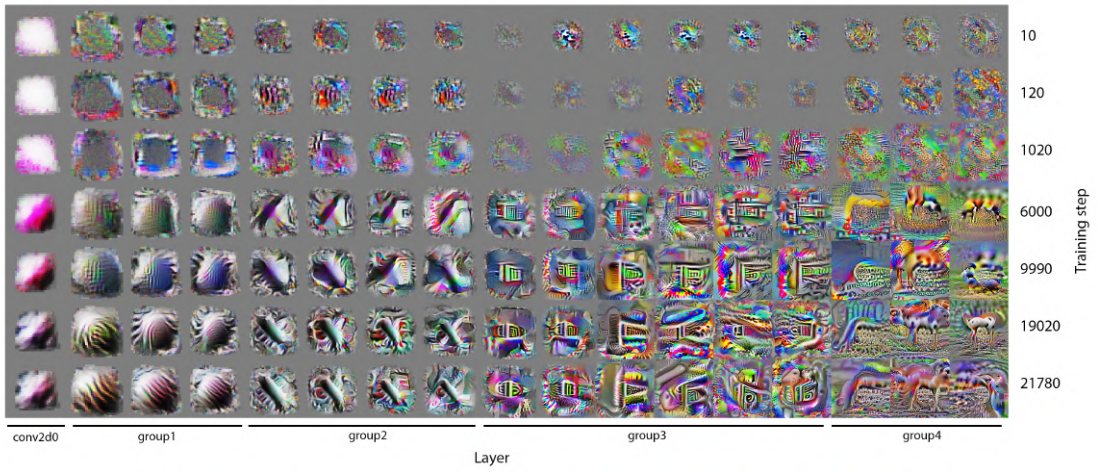


Figure 4.1: Convolution kernels of 6 different channels (out of 64) from the first conv layer of ResNet 18. Each 7×7 square is one kernel. Groups separated by horizontal white space each correspond to one of the 6 channels. Within-group rows correspond to different adversarial epsilon values. Columns signify training steps.



(a) Feature visualizations for various steps during standard training, rows correspond to training steps 20, 110, 1150, 6160, 9560, 19000, 21780.



(b) Feature visualizations for various steps during adversarial training, rows correspond to training steps 10, 120, 1020, 6000, 9990, 19020, 21780.

Figure 4.2: Comparison of feature visualizations during different training methods. Feature visualizations are shown for Neuron 0 of several layers from ResNet 50. They have been generated without Fourier parameterization, and they are cropped to the approximate receptive field of the respective layer. Early layers are on the left side, later layers are on the right side.

4.1.3 The Influence of the Attack Epsilon on the Feature Representation

In Section 4.1.2 we compared feature visualizations during standard training (adversarial perturbation epsilon of 0) with feature visualizations during adversarial training (using adversarial perturbation epsilon of 1.0). To further investigate the role of the epsilon value, we trained a series of models adversarially with increasing epsilon values. For this experiment we chose a ResNet 18, which is significantly faster to train when compared to the much deeper ResNet 50, but otherwise shares many of the larger model's architectural properties. We trained on Restricted ImageNet with batch size 256 for 30K training steps. Then, we generated feature visualizations for Neuron 0 of the first convolutional layer and the final ReLU-layers of each of the eight residual blocks. The result is shown in Figure 4.3, where each row corresponds to one epsilon value and each column corresponds to one layer.

We can see a strong difference regarding spatial image frequency between feature visualizations of the last layer (right-most column in Figure 4.3). The further we look left towards earlier layers, the less we can spot a perceivable difference in image frequency between epsilon values. We can also observe that the frequency difference between 0.33 (third row from bottom) and 1.0 (bottom row) is hardly perceivable, suggesting a saturation effect of the phenomenon. This saturation seems to happen at even lower epsilon values in earlier layers, although a meaningful visual comparison is prevented by the lack of alignment of features between the models.

4.2 Adversarial Transfer Learning

The experiments we described so far all involved training from scratch, meaning they all started from a fixed-seed randomly initialized model. We learned that adversarial training diverges early during training and converges to a completely different feature representation than standard training, even with small perturbation epsilons.

As mentioned in the previous section, the non-alignment between features makes it hard to compare features learnt by individual neurons. The (fixed-seed) random weight initialization we used to train our models does not give the optimization enough direction to enforce a consistent feature representation across the hyper-parameters we vary in our experiments.

This leads us to investigate possibilities for training CNNs adversarially in a feature-preserving way. Transfer learning, as explained in Section 2.1.3, is a widely used technique among deep learning practitioners for adapting a pre-trained model to a specific task. For our purpose, "adversarial transfer learning" turns out to preserve the model's feature representation well enough to enable visual re-identification of features across all convolutional layers after training. We show examples of feature visualizations before and after adversarial transfer learning in Figure 4.4.

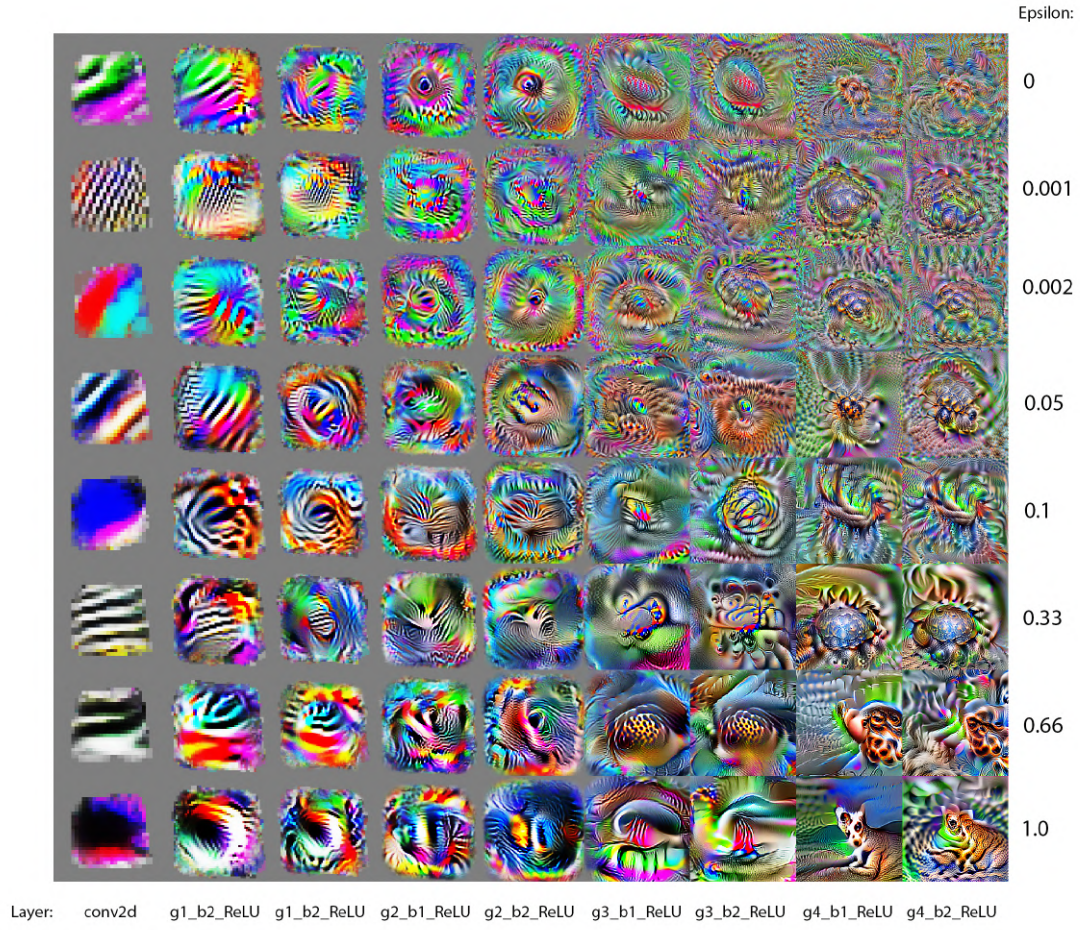


Figure 4.3: Feature visualizations (without Fourier parameterization) of Neuron 0 of several layers (from left to right) from ResNet 18. Each row corresponds to one L_2 epsilon value used for adversarial training. From top to bottom: 0 (std. training), 0.001, 0.002, 0.05, 0.1, 0.33, 0.66, 1.0.

It is important though to make a distinction to commonly used transfer learning here. Usually, transfer learning is used to either take advantage of the better generalization capabilities of a network that has been pre-trained on a larger dataset before being fine-tuned to a smaller dataset, or to save computational resources when a complete training run would be too expensive. In our case on the other hand, transfer-learning is done on a variation of *the original dataset*. This allows us to specifically investigate the effect of the type of variation we use to modify the original dataset. Allen-Zhu and Li have investigated adversarial transfer learning [AZL20]. They provide detailed mathematical explanations and proofs for a phenomenon they describe as "feature purification". We take a more visual approach and generate fine-grained progressions of feature visualization along the training. Szabo et al. [SKC+20] showed evolving feature visualizations, similarly to us, although during "traditional" transfer learning from one dataset to another. One example from their paper is transfer learning from ImageNet to Places2 [ZLK+17].

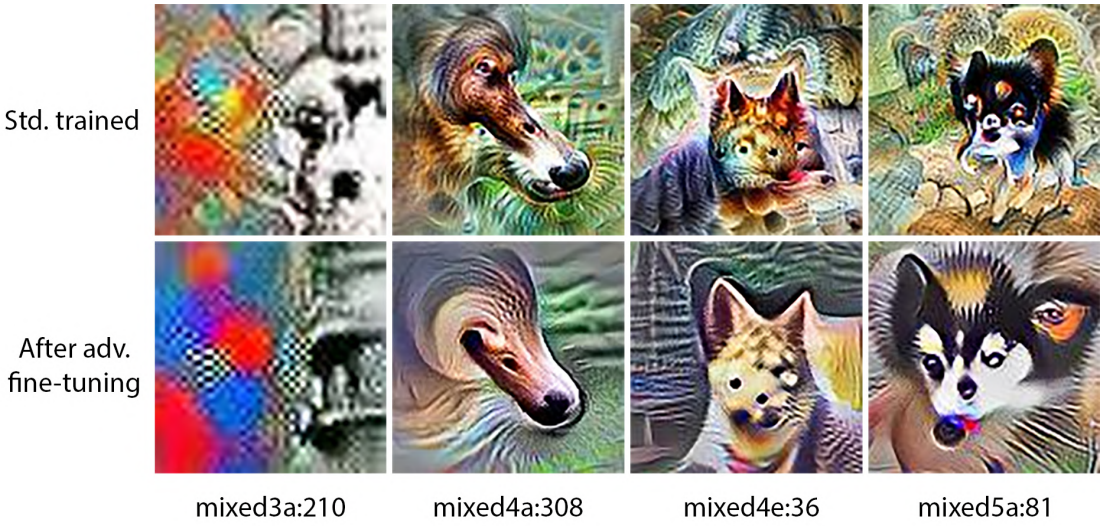


Figure 4.4: Examples of single-neuron feature visualizations ("neuron" optimization objective in *Lucid*) from Inception V1. Visual similarity is encountered at neurons in all layers, although later layers tend to diverge at a higher rate. The "mixed5a" example had to be cherry-picked, as many neurons do not have such a strong visual similarity before and after adversarial transfer learning. In early layers, most neurons are easy to recognize after transfer learning, but often minor semantic changes are visible (for example color shift in "mixed3a:210").

4.2.1 Adversarial Transfer Learning Preserves Feature Representation

We started our investigation of adversarial transfer learning on ResNet 18. We took a model that had been trained on Restricted ImageNet with standard training for 33K training steps and trained it adversarially on the same data. We saved checkpoints every 10 training steps during training. Figure 4.5 shows the feature visualizations for the

third neuron of 9 layers during training steps 0,10,20,40,200, 500, 1000, 6000, 12000 of adversarial fine-tuning. The feature visualizations have been generated without Fourier parameterization, a technique often used for generating more pleasing images. We thus avoid biasing the visualizations towards a more evenly distributed frequency spectrum. We can see that the features in all layers preserve their high-level appearance while becoming significantly richer in low-frequency information.

While the increase of low-frequency content is to be expected from the discoveries by Tsipras et al. [TSE+19], the fact that the visual feature semantics get mostly preserved by adversarial transfer learning, is a non-trivial discovery. This in turn enables us to directly compare feature visualizations and weights from standard- and adversarial models against each other.

4.3 Adversarial Transfer Learning with Inception V1

Our experiments with ResNet 18 and Restricted ImageNet showed that with this model/dataset combination, adversarial transfer learning preserves the feature representation to a very high degree. As this network is comparatively shallow and the dataset is very simple in terms of the number of classes, those results do not necessarily generalize. To test this behavior on a deeper model and a larger dataset, but also to be able to take advantage of existing interpretability research such as the Circuits project, we repeated the adversarial transfer learning experiment with Inception V1 and (non-restricted) ImageNet. For this, we extracted the weights from the trained Inception V1 model accessible through the Lucid feature visualization library.

During this experiment, we encountered an interesting phenomenon that we did not observe with ResNet 18 / Restricted ImageNet. The error rate stayed roughly constant at 1.0 during the first 10K training steps, then steeply dropped to 0.6 top 5 training error at around training step 15K, and then continued to decrease more slowly. To analyze this further, we estimated activations from the validation set for multiple steps during training and correlated them with the activations at the initialized state (step 0). The activations were estimated from a random subset of 512 images from the validation set. As an example, for layer “conv2d0” with an output resolution of 112×112 and 64 output channels, this would result in an activation tensor of shape $512 \times 112 \times 112 \times 64$. We then combined the first three dimensions and sampled 20K values from the combined dimension, resulting in a tensor of shape 20.000×64 for our “conv2d” example. This tensor was then flattened to be correlated with an identically calculated (and identically sampled) vector from another training step. We plotted these correlations, separated by layer, in Figure 4.6.

Similarly, we compared the learned weights per layer over the training by correlating them with their initial state (Figure 4.7). We can see that during adversarial transfer learning, all but the first three layers experience a significant intermediate drop of activation correlation, before recovering around training step 15K. This recovery happens approximately between training steps 10K and 15K, which is also where the error rate

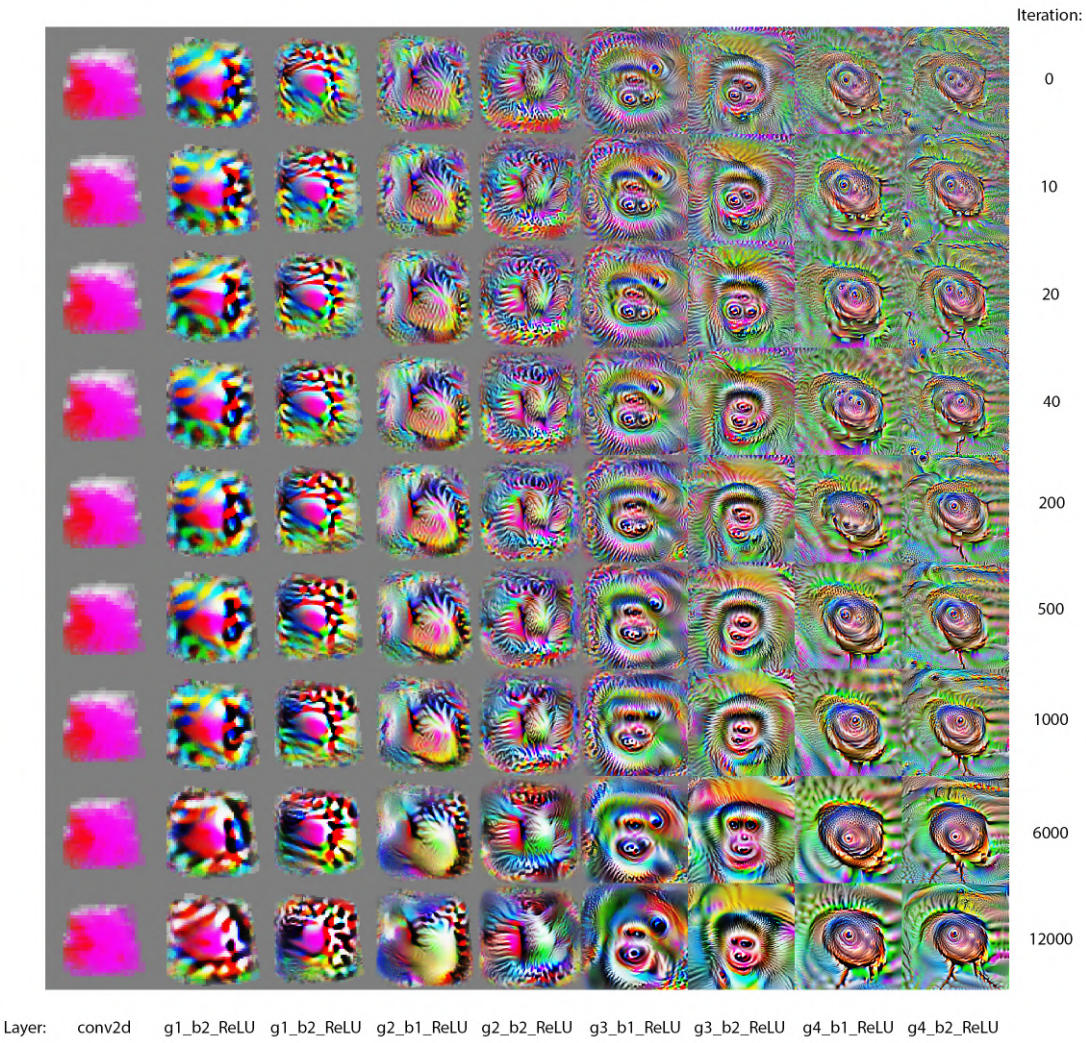


Figure 4.5: Feature visualizations (without Fourier parameterization, cropped to approximate receptive field) of Neuron 2 of several layers (early layers left to later layers right) from ResNet 18 during adversarial transfer learning starting from a standard-pre-trained network. Training steps: 0,10,20,40,200, 500, 1000, 6000, 12000.

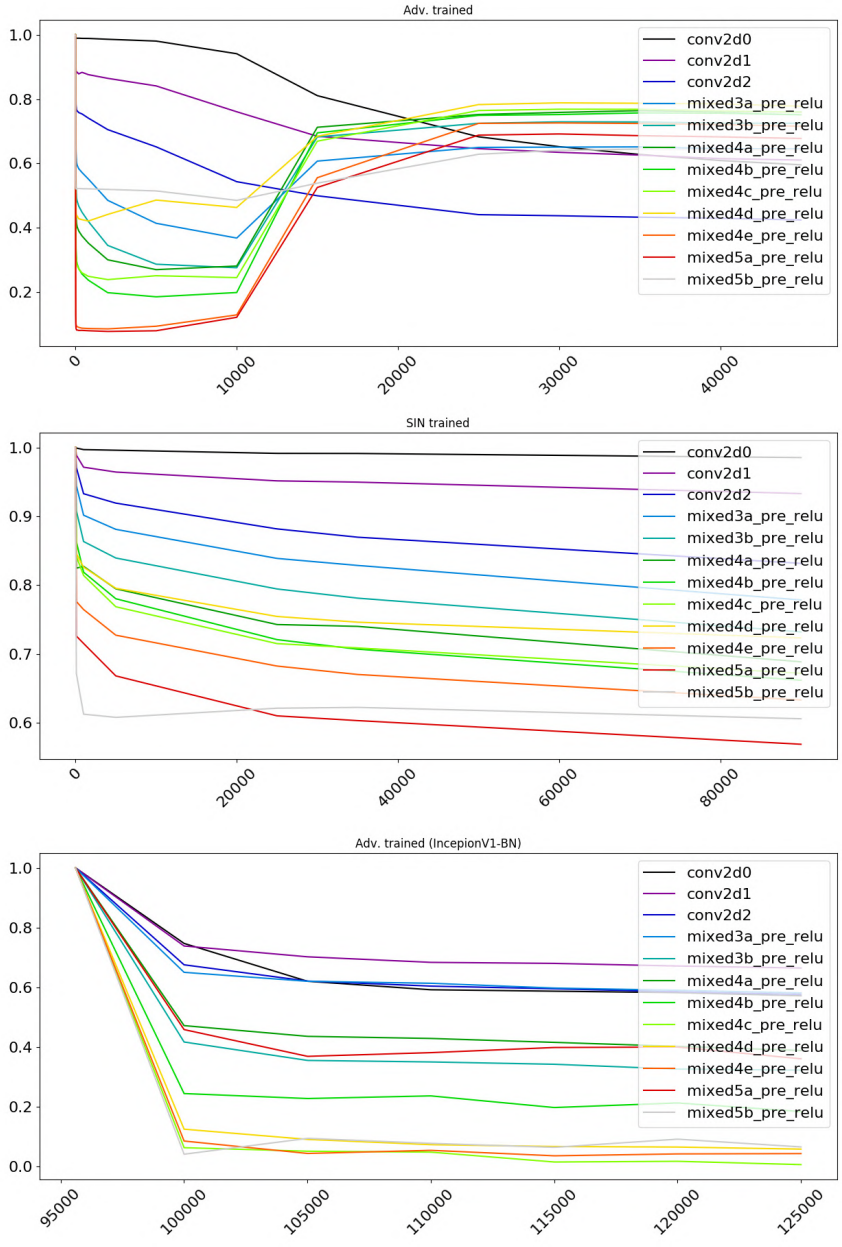


Figure 4.6: Activation correlations: Activations over the ImageNet validation set for multiple steps during training are compared with the activations at the initialized state. For "mixed" blocks, the concatenated activations of all sub branches before the ReLU layer have been evaluated ("pre_relu"). X-axis represents fine-tuning iteration, y-axis represents correlation coefficient. Refer to the main text for a detailed explanation.

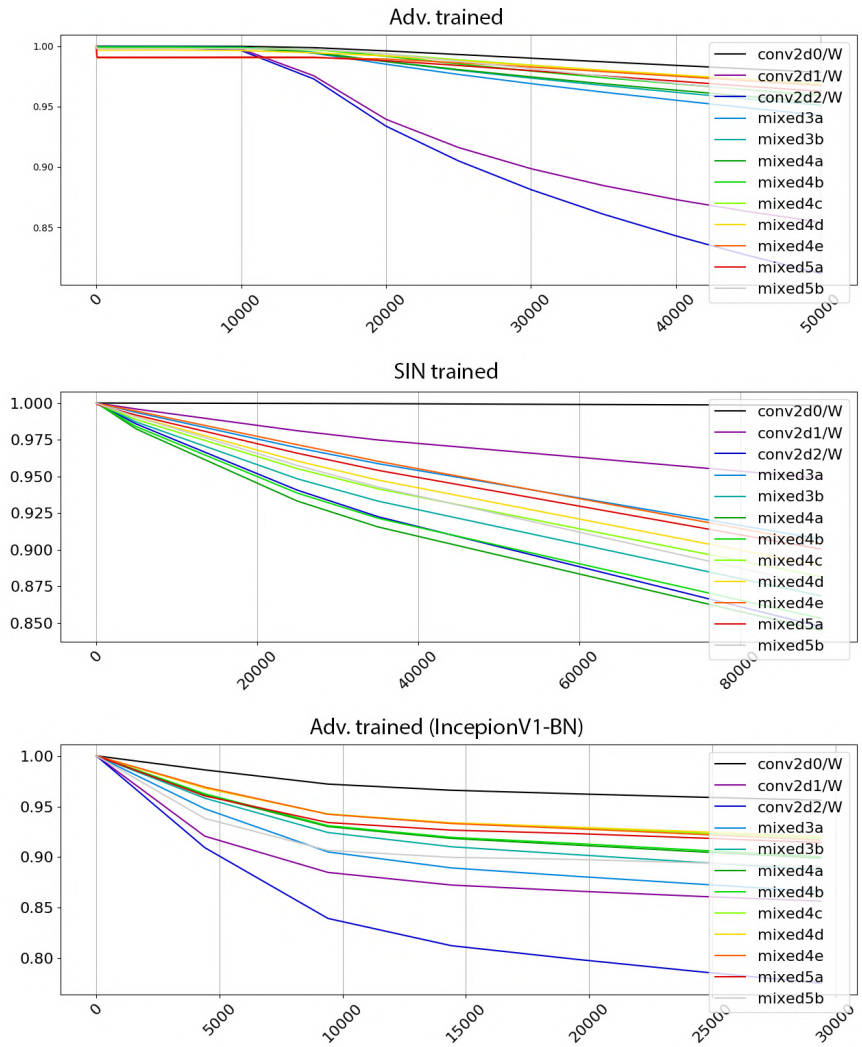


Figure 4.7: Weight correlations: Correlation coefficients (y-axis) of learned weights from each layer correlated with their initial state. X-axis represents fine-tuning iteration. Coefficients for all layers within a "mixed" block have been summarized by their mean. Refer to the main text for a detailed explanation.

decreased most significantly during training (Figure 4.8). The weight correlation stays high for all layers until around training step 10K, after which "conv2d1" and "conv2d2" start to change gradually. After around training step 15K the other layers' weights also start to change, but less than "conv2d1" and "conv2d2".

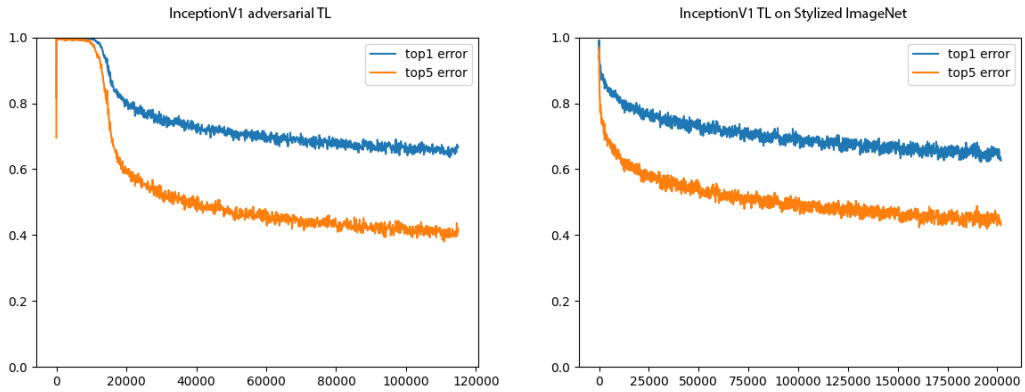


Figure 4.8: Left: Training error during adversarial transfer learning on ImageNet. Right: Training error during transfer learning on Stylized ImageNet.

This phenomenon seems to be only caused by *adversarial* fine-tuning, as our experiments with Stylized ImageNet did not result in similar behavior (Figure 4.8 right, middle of Figures 4.6 and 4.7). We hypothesized that the fact that Inception V1 has no batch normalization (BatchNorm / BN) [IS15] layers is responsible for this phenomenon. We designed an experiment to provide further evidence for this hypothesis. We constructed a new model with a batch normalization layer after each convolutional layer and after each fully connected layer. We initialized the weights of this new model with the Inception V1 weights and trained for approximately 95K training steps until there was very slow training progress (error curves shown in Figure 4.9 left). Then, we fine-tuned the resulting model adversarially for more than 30K iterations. We observed that the error rate had stopped decreasing even earlier, as seen in the error rate plot in Figure 4.9 right. We generated feature visualizations (Figure 4.10 bottom) and plotted activation correlations (Figure 4.6 bottom), as well as weight correlations (Figure 4.7 bottom) throughout the fine-tuning process. We can see that the phenomenon with the initial error-rate-plateau and subsequent rapid drop that we encountered in the unaltered Inception V1 adversarial fine-tuning is not observable in the BatchNorm-augmented version. Also, activation and weight correlation plots for this training run look more regular.

We further made multiple models at important steps during training available in our interactive application, enabling the detailed inspection of model behavior (including activations, feature visualization and classification output).

4.4 Training with Stylized ImageNet

Madry et al. [MMS+18] provided evidence that PGD is the strongest first-order attack within an L_p-bounded epsilon ball. Lifting the L_p-bounding constraint naturally opens up the space of allowed image transformations to a large variety of clearly human-visible perturbations. One such method is style transfer, which strongly distorts image texture while keeping the overall structure and shape largely intact. Geirhos et al. [GRM+18] showed that training on images with changed artistic style leads to a model that focuses more on shape than on texture for its predictions. This can also be seen as a form of robust training, forcing the model to learn a global structure rather than high-frequency texture features.

To evaluate whether the low-frequency feature visualizations encountered in adversarially trained (or fine-tuned) models are a unique phenomenon of adversarial training, we generated feature visualizations for a model that has been fine-tuned on Stylized ImageNet, starting from the same Inception V1 used as a base for our adversarially fine-tuned model. As an example, we can have a look at the "dog head detector"-neuron "mixed4a:222", inspired by Olah et al. [OCS+20]. Over the course of the training, we can observe a slight change in shape and color, but the overall "style" and frequency distribution is very similar to the starting point (Figure 4.11). This experiment highlights the unique properties that adversarial training induces on the learned features in CNNs. We made several checkpoints of this fine-tuning experiment available in Perturber for detailed inspection.

4.5 Conclusion of Initial Experiments

In our preliminary experiments, we explored various aspects of adversarial training:

- We showed the highly non-linear nature of adversarial training by comparing learned features between CNNs with identical initialization and training schedule, but slightly different perturbation epsilon.
- We proposed transfer learning as a feature-preserving method to obtain an adversarially trained model that can be directly compared to a base standard model.
- We visualized the adversarial fine-tuning of ResNet 18 and Inception V1, found an interesting behavior (features disappearing temporarily) in the latter, and provided results supporting the hypothesis that it is caused by the absence of Batch Normalization layers.
- Finally, we provided evidence that the low frequency, "human-aligned" feature visualizations obtained from adversarially trained models are a characteristic phenomenon by showing that training on Stylized ImageNet does not result in similar feature visualizations.

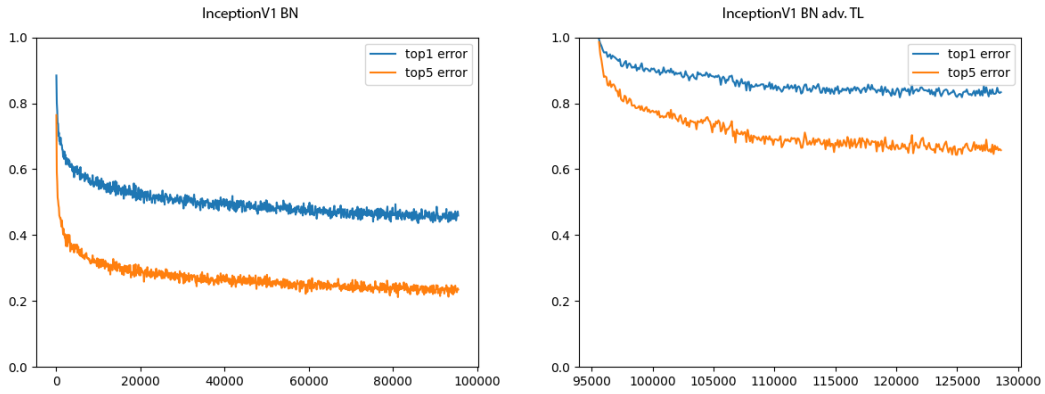


Figure 4.9: Left: Training error during re-training Inception V1 on ImageNet after inserting BN layers. Right: Training error during adversarially fine-tuning the resulting model (also on ImageNet).

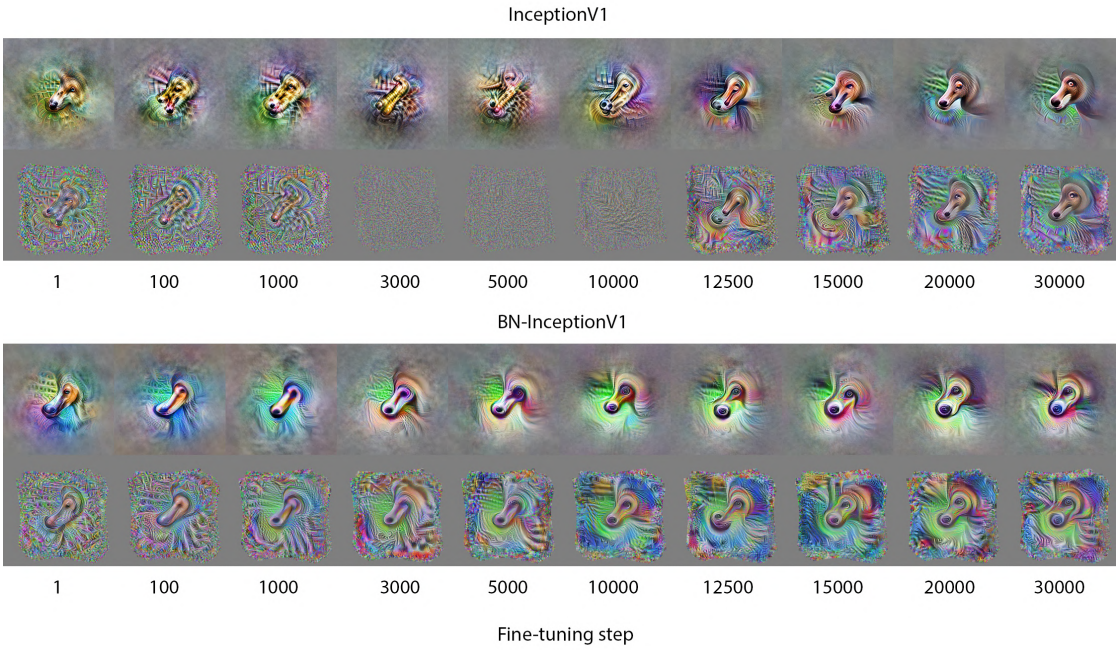


Figure 4.10: Feature visualizations of dog head detector throughout adversarial fine-tuning for Inception V1 (top) and BN-modified Inception V1 (bottom). Top rows are parameterized in Fourier space and with decorrelated colors, bottom rows do not use these parameterization tricks and are directly parameterized as pixel images. It can be seen that the BN-modified model does not exhibit the phenomenon where intermediate checkpoints hardly allow any gradient to flow back to the input, preventing meaningful feature visualizations without parameterization tricks. Even with the parameterization tricks, the BN-modified version has a more gradual change of feature visualizations.

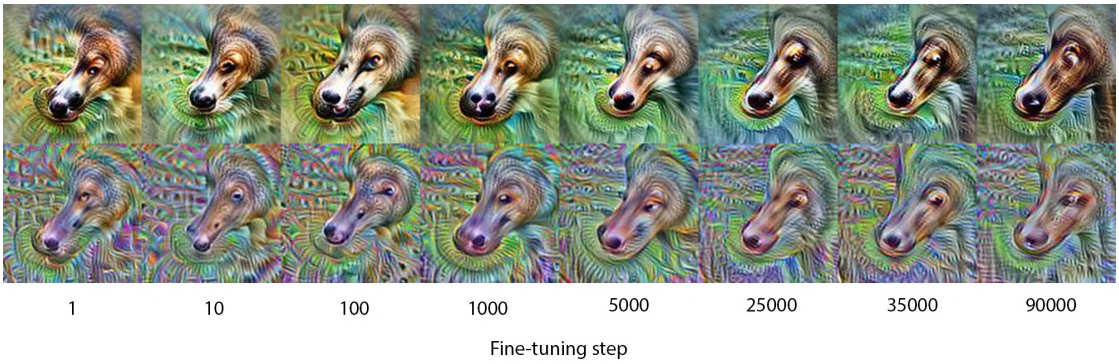


Figure 4.11: Feature visualizations of dog head detector throughout fine-tuning on Stylized ImageNet. Top rows are parameterized in Fourier space and with decorrelated colors, bottom rows are directly parameterized as pixel images. While the feature's appearance changes, the spatial frequency distribution does not change significantly when compared to adversarial fine-tuning (Figure 4.10).

With respect to our first hyposthesis (H1), our results strongly highlight the distinctive role of adversarial training. Increasing the magnitude of the adversarial perturbations during training strongly correlates with the magnitude of low frequencies in feature visualizations. For adversarial transfer-learning on Inception V1, we identified a clear relation of the appearance of feature visualizations to the training progress in terms of error rate. On the other hand, feature visualizations were not able to visually explain the difference of training on Stylized ImageNet when compared to training on standard ImageNet. H1 states that feature visualization can help explain the difference between standard training and *robust training* in general. Our results on the other hand show that feature visualizations clearly benefit the understanding of adversarial training, but not necessarily the understanding of other robust training methods, training on Stylized ImageNet in our case.

CHAPTER 5

Visual Analytics Design of Perturber

Perturber aims to provide users an interactive playground where CNN output and intermediate activations can be observed in response to input image changes, with the ability to compare models with each other. Our hypothesis H2 states that by being able to interactively manipulate a synthetic input scene with a large and diverse set of parameters, and observing the changing activations instantly, researchers can quickly generate and then confirm or reject hypotheses. By immediately observing effects of changes, they can be more efficient in building a strong intuition for the highly complex behavior of convolutional neural networks. Perturber is designed to facilitate the exploration of robustness and potential vulnerabilities. This is reflected in the types of the provided input perturbations and in the models provided for comparison: A standard trained model and two robustly fine-tuned variants.

We formalize these aspects into six core requirements:

- **R1**: Rich input perturbations. The application should provide an extensive set of tools to manipulate and perturb a synthetic input scene with fine-grained control. These tools should be able to effectively probe the robustness of the models.
- **R2**: Interactivity. The application should be highly interactive and allow the user to quickly explore CNN responses to input perturbations with instant feedback. Complex visualizations which cannot be rendered in real-time should be avoided.
- **R3**: Visualize output and hidden layers. To be able to form hypotheses about model behaviour, users should be able to view the output of a model, but also the activations in hidden layers. Hidden layers' activations contain information *how* a model derived its output, and their understanding is one of the great challenges in

ML interpretability research. Users should be able to build intuition by exploring their responses under to a changing input scene.

- **R4**: Model comparison. Robust training methods result in models that are more robust to targeted vulnerabilities. The application should let the user directly compare robustly trained models to a standard trained model, to let them explore *how* and *where* they behave differently.

- **R5**: Intermediate Checkpoints. In order to investigate the models during training, the application should provide intermediate checkpoints at multiple stages during training, including respective feature visualizations.

- **R6**: Model editing. To let the user explore how compatible the learned weights of the fine-tuned model variants are, with each other and with the standard trained base model, there should be a mechanism to mix their weights on a layer-by-layer basis.

For the design of the application, we build upon our findings presented in Chapter 4: To facilitate model comparison, we take advantage of feature alignment after transfer learning. We incorporate multiple checkpoints during adversarial transfer learning, with the checkpoint indices chosen to allow investigating the abnormal transfer learning behaviour presented in Section 4.3. To allow further investigation of the findings from Section 4.4, we provide both an adversarially fine-tuned model, as well as a model fine-tuned on Stylized ImageNet in the Perturber application. Together with the standard model, three models are provided. Apart from our own findings, we facilitate the investigation of phenomena presented by others, such as the texture-shape cue conflict presented by Gheiros et al. [GRM+18]. We will look at them in detail in Section 5.2.

The user interface of Perturber can be seen as a concise distillation of the most task-relevant components from a larger set of building blocks that we had experimented with during development. In order to design a feature-rich application without sacrificing usability, we showed prototype components to domain experts and discussed the components' usefulness with them. While the Scene View, responsible for input generation, could be directly transferred to the Perturber interface, we chose to abandon some of the more complex components. For example, we had developed a module to generate two-dimensional tuning curves, similar to those found in the OpenAI Microscope [Ope]. They are shown in Figure 5.1. Apart from showing the tuning curve plots, the module could then compute and show clusters of neurons from a selected layer, allowing the discovery of neurons with similar responses to the two varied parameters. During discussions with our domain experts however, we found that the generation of tuning curves was not in line with the goal of maximizing interactivity. Although generating the tuning curve plots only took on the order of ten seconds, we decided that there was no place for such functionality in the otherwise highly responsive Perturber application. Additionally, instead of confronting the user with a more versatile but overwhelmingly

complex interface, we preferred to focus the user's attention to a set of powerful core interface components.

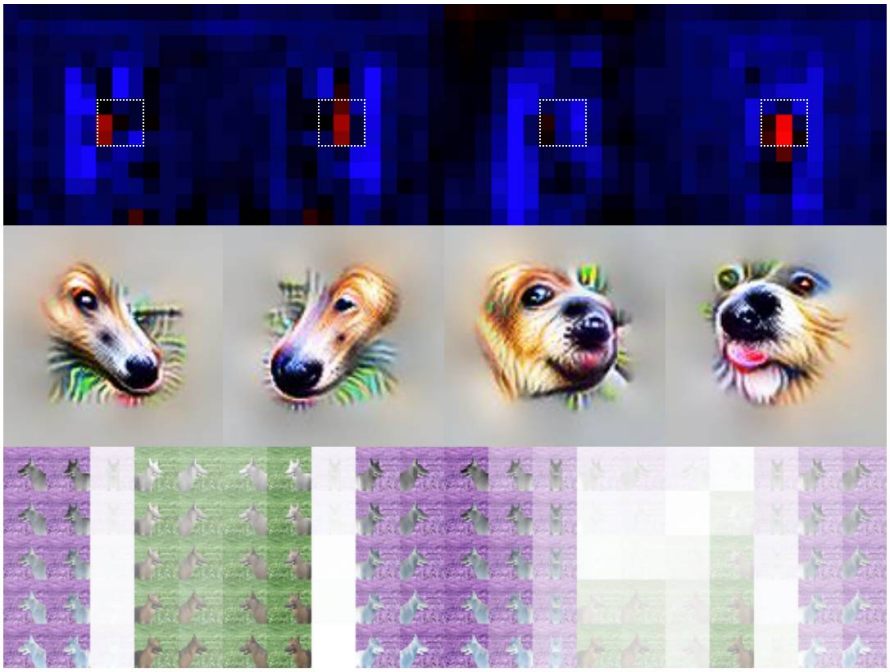


Figure 5.1: One of the components that we did not incorporate into the final version of Perturber: A module for generating two-dimensional tuning curves. The 2d tuning curve plots can be seen in the bottom row of the screenshot. The yaw-rotation of the dog, as well as the texture influence have been varied. Each thumbnail shows the input and is weighted according to the response of the respective neuron. Negative response leads to a color-inverted thumbnail (purple here). The first column from left shows the "right-facing dog head detector". The corresponding tuning curve plot at the bottom clearly shows positive responses for the right-facing dog inputs and negative responses for the left-facing dog inputs, regardless of texture strength.

5.1 Overview

The general concept of Perturber is to provide an environment where the user can interactively inspect model behaviour. We identify three main conceptual components that can be used as standalone tools or in any combination between them.

- **Interactive Exploration.** Perturber is built around a 3d scene that can be interactively manipulated in numerous ways. A 3d scene gives the user fine, repeatable control over all aspects of the input image and lets them systematically and independently perturb the scene parameters. This makes it preferable over other potential input sources such as a webcam feed. Simultaneously to the user

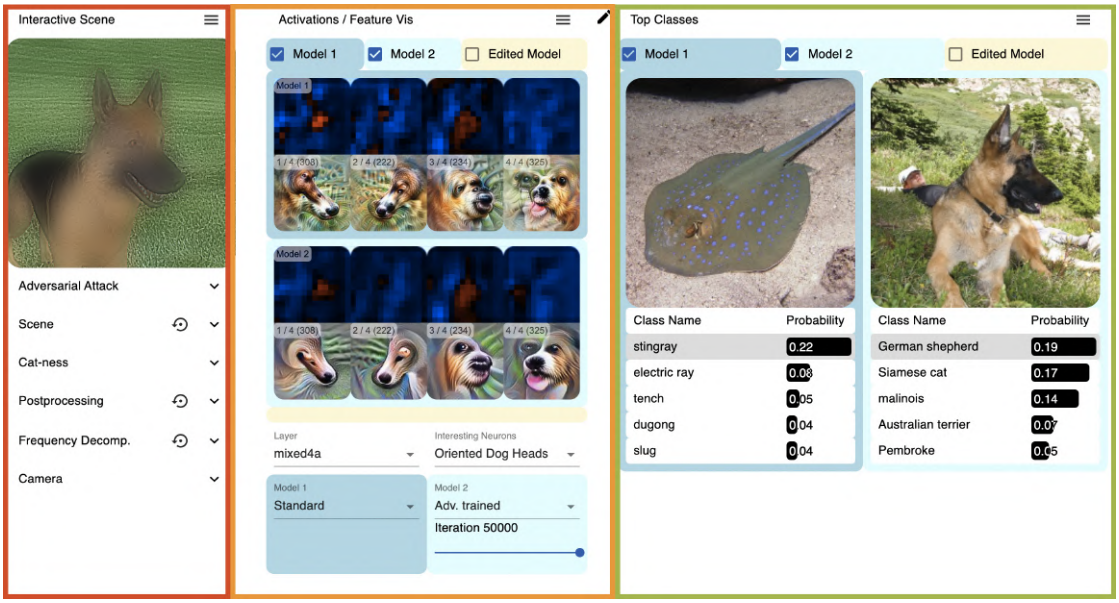


Figure 5.2: Input / Scene View (red); Intermediate layers / Model View (orange); Output / Prediction View (green).

manipulating the input scene, the activations at selected model stages (intermediate layers, output layers) are displayed according to the current input. In intermediate layers, these activations are 2d activation maps that can be displayed for selected channels. Additionally, the top 5 predictions can be displayed in a bar-chart, with dataset examples used to visualize class labels.

- **Model Comparison.** The user can choose two models out of the provided three models to compare to each other. They are assigned to two slots in the interface and subsequently denoted *Model 1* and *Model 2*. Respective visualization interface components are colored with two shades of blue, Model 1 corresponding to the darker shade and Model 2 to the lighter one (as seen in Figure 5.2). The provided models are based on ImageNet-trained Inception V1, with two fine-tuned versions and intermediate checkpoints. As we discussed in Section 4.2, fine-tuning strongly preserves feature representation, which makes direct per-neuron model comparison possible. The user can compare activation maps and predicted class probabilities in response to interactive scene perturbations, as well as pre-calculated feature visualizations for various layers within the model and for various checkpoints along the training process.

- **Model Editing.** Perturber provides tools for editing a model by either mixing the weights between two models or by pruning the weights by kernel magnitude. The edited model can be used in the above described exploration concepts. It can be compared to either of the two base models by activation as well as by feature visualizations, which can be generated for the edited model at interactive speed.

The edited model can also be used to generate adversarial attacks for the input scene. These can be seen as highly experimental features. We did not encounter any existing tool that allows interactive weight editing, but we were able to make interesting discoveries with it. We discuss them in Chapter 7.

The user interface has three main components that are arranged in columns. This three-column arrangement symbolizes the data flow through the network, from input (left) to prediction (right), as shown in Figure 5.2.

Starting on the left side, the Scene View (Section 5.2) serves as an interface to control the generated input image by manipulating the 3d scene as well as 2d post processing parameters. In the middle column, the specific models, which vary by training data and iteration, can be selected and the activations of intermediate / hidden layers can be inspected and compared. Also, the user will find the weight editing interface here, a component that lets the user interpolate weights between Model 1 and Model 2, or prune weights by magnitude. The resulting model, the *Edited Model*, is the third model for which the various outputs can be visualized. On the right side, the classification logits of each model can be viewed and compared among them.

5.2 Scene View

The Scene View gives the user control over the input image, serving as the base for the exploratory analysis. It contains an interactive 3d scene that can be manipulated in numerous ways that are aimed at testing the robustness of the models. The rendered image of the 3d scene is resized to the model's input size (224×224) and subsequently used for inference to generate activation data interactively.

There are various examples in literature, where *offline* experiments are performed that involve perturbing the input image of a CNN and measuring its activation change in response to the input change:

- **Synthetic Tuning Curves (Figures 5.3 and 5.4):** This is a technique which has its origins in neuroscience [BG06] and can be found extensively in the Circuits project as well as on OpenAI Microscope [Ope]. Various simple two-parameter synthetic image generation processes are grid-sampled over both parameters (for example combinations of frequency, orientation, curve radius, gradient width, color hue, etc.) and the response is plotted in a respective 2d grid. In Figure 3.7 at the top, for example, we can see that the selected neuron responds more to slanted patterns than to strictly axis aligned patterns.
- **Error-rate heat map in response to frequency perturbation (Figure 5.5):** This is an example by Yin et al. 2019 [YLS+19], where a 2d parameter space is grid-sampled in the Fourier domain. The resulting perturbation vectors are sine-waves in varying directions and frequencies. They are used to investigate

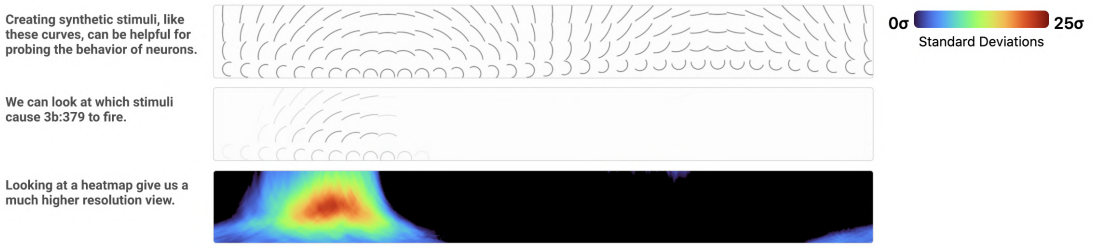


Figure 5.3: Example taken from the "Curve Detectors" article by Cammarata et al. [CGC+20]. The authors explore the response of a specific neuron in a CNN by varying synthetically generated curve images over combinations of orientation and radius (top). The resulting response is plotted as a heatmap (bottom), additionally they show a figure where the input image is weighted by the response (middle).

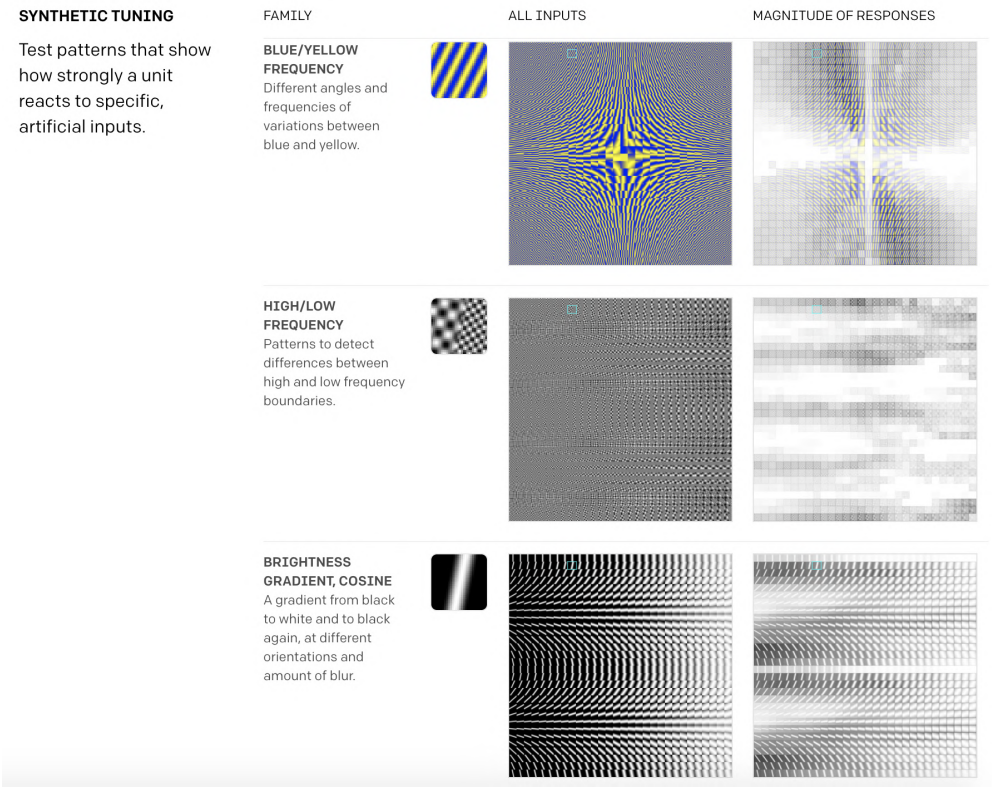


Figure 5.4: The same concept as the one shown in Figure 5.3 is used in OpenAI Microscope [Ope], like shown in this screenshot. A pre-defined set of synthetic input images it fed into the CNN and the response is plotted for practically each neuron in the model. All input images are generated by 2 independently varying parameters.

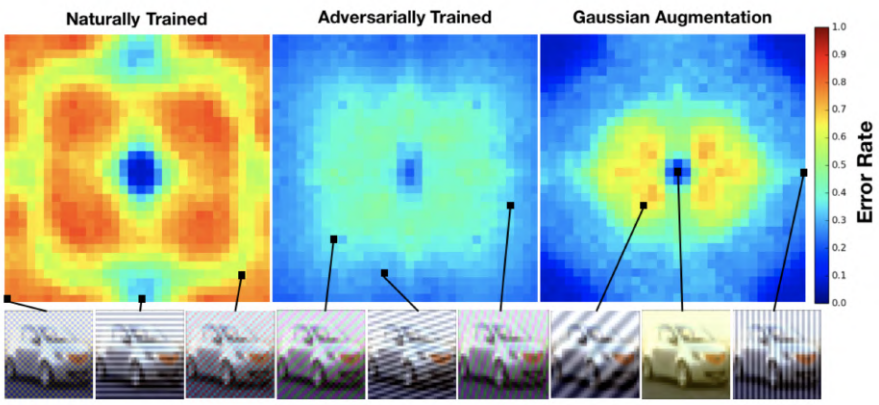


Figure 5.5: Figure taken from Yin et al. [YLS+19]. The authors use a 2d perturbation space in the Fourier domain (direction and frequency) to investigate multiple model's (error rate-) sensitivity regarding noise of various frequencies. Examples of perturbed images are shown in the bottom row, the heat maps show the error rate over the 2d perturbation space.

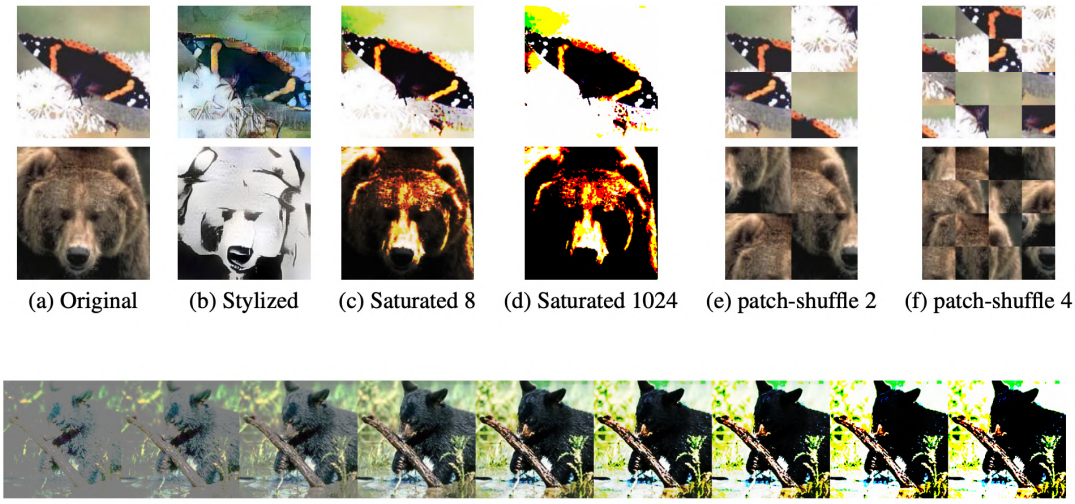


Figure 5.6: Figure taken from Zhang et al. [ZZ19], where multiple model's error rates are compared in response to perturbations that either change the texture while preserving shape (top, b - d) or destroy the shape while preserving texture (top, e and f). An example parameter progression is depicted for "Saturation" in the bottom part of the figure.

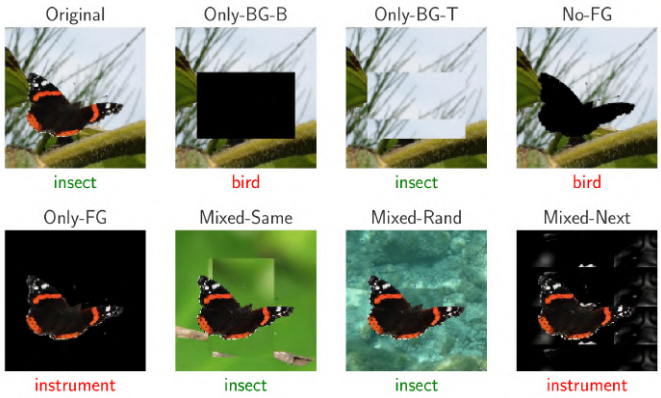


Figure 5.7: The test cases from the *Backgrounds Challenge* [XEIM20]. In the top row, the three cases *without* the foreground object are depicted, where the model can only rely on the background or the foreground silhouette to make its prediction. The bottom row shows the four cases with *only* the foreground object. Figure taken from Xiao et al. [XEIM20],

multiple model's robustness in different frequency bands. The results from that work show that adversarially trained networks are very sensitive to low-frequency perturbations but robust to high-frequency perturbations, whereas for standard trained models it is the other way around.

- **Shape- and texture-changing image perturbations (Figure 5.6):** Zhang and Zhu [ZZ19] investigate a set of image perturbations individually regarding multiple model's error rate change in response to perturbing the test set. They differentiate between perturbations that destroy texture (at some magnitude) while preserving shape (style transfer, saturation change), and perturbations that preserve texture while destroying shape (Patch Shuffling). Most experiments in this work can be seen as a one-dimensional grid-sampling. Among other findings, they show that adversarially trained models are more sensitive to shape-destroying perturbations like patch-shuffling when compared to standard trained models, whereas standard trained models are more sensitive to texture-destroying perturbations, like saturation or contrast changes.

- **Exchange of backgrounds (Figure 5.7):** Xiao et al. [XEIM20] investigate the reliance of image classifiers on backgrounds. They introduce seven different test variants, along with respective datasets, to facilitate their analysis. They find a significant accuracy drop when swapping the background of an image at test time with a background from another random class (Figure 5.7 "Mixed-Rand").

All of the above-listed experiments involve automated offline computations where a well-defined parameter space is grid-sampled before visualizing the result. They thoroughly

investigated the respective networks' behaviour to isolated perturbations, but they are inherently limited in the possible combinations of parameter changes, as full grid-based exploration of large parameter spaces is intractible. Also, any grid-sampling of a parameter space is hard to visualize for more than two parameters. While a 2d heatmap is highly readable, a number of parameters that goes beyond two can not be plotted effectively. Additionally, most of these experiments either work well for early layers [CGC+20, Ope] or output layers [YLS+19, AAB+15, XEIM20], while the intermediate layers with more complex features are not being investigated [YLS+19, ZZ19, XEIM20] or would not be useful as the later layers' features are far more complex than the input images designed for the respective experiment [CGC+20, Ope]. Intermediate layers often detect high-level object parts like dog-heads, snouts, ears, tyres, car-windows etc. instead of low-level shapes like curves or texture patterns. Using a 3d scene lets us generate fine-grained variations of the input image, benefiting the investigation of units in intermediate layers. We can probe them by varying the camera, rendering and post-processing parameters, just like early layer units are probed by varying primitive shapes in OpenAI Microscope [Ope] and by Cammarata et al. [CGC+20]. As our application is designed to be used interactively, the user can intuitively direct the parameter combinations they want to explore, instead of relying on inefficient high-dimensional grid-sampling.

5.2.1 3d Scene

Our primary 3d scene is designed to activate a large number of neurons in our ImageNet-trained Inception V1 model. As almost 10% of ImageNet's classes are dog breeds, many neurons of our CNN respond to dog-related features [OMS17]. The choice of a dog as our primary scene's foreground object is therefore straightforward. We complement the dog foreground object with a background image of a lawn. In order to expand the exploration capabilities of the scene, we allow seamlessly morphing of the dog into a cat. Likewise, we provide a secondary object pair of a race car and a fire truck that can be morphed into each other. This second pair has been chosen to strongly contrast the natural organic shapes of the animal pair with human-made, hard-surfaced machines. A secondary background image of a street scene complements the race car / fire truck pair, but the user is not restricted to these "matching" foreground / background combinations. They can also test the model's response to cue conflicts such as the race car in front of the lawn background. Furthermore, the users can choose custom 3d objects and background images from their local file system. The 3d file has to be in *Wavefront OBJ* (.obj) format. Separate textures for diffuse color and for the alpha channel can be uploaded along with the .obj file. The top two rows of Figure 5.8 show the morphing transitions from dog to cat, and from fire truck to race car in front of their respective backgrounds.

The view can be manipulated by mouse interaction. Dragging within the scene area *rotates* the camera along pitch- and yaw axes (but keeping the model centered in view), scrolling *dollies* the camera. Middle-mouse dragging *pans* the view.

Beneath the 3d scene area, there are sliders for controlling the scene parameters, as well as controls for performing adversarial attacks. To improve usability, they are arranged

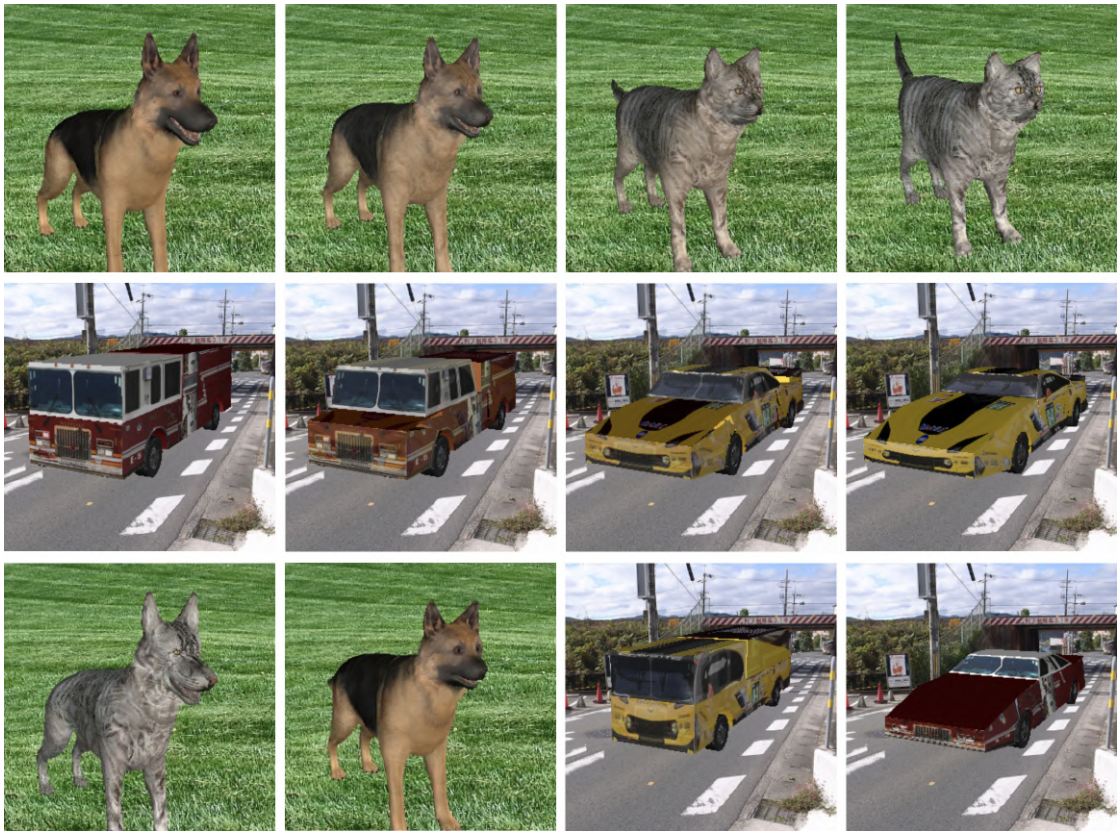


Figure 5.8: The top two rows show the morphing transitions from dog to cat, and from fire truck to race car. The bottom row shows disentangled settings for shape and texture morphing.

into semantically coherent groups. These controls and the reasoning behind including them into the interface are described below in more detail. In total, Perturber provides 19 continuous input perturbation parameters, in addition to tools for generating adversarial attacks, controlling the camera rotation, and changing foreground object or background image.

5.2.2 Parameters

As understanding CNN robustness is the primary motivation behind the Perturber application, the choice of parameters is designed to help with the testing of CNN's robustness. Besides the basic camera manipulation tools, some of the parameters facilitate the disentanglement of shape and texture cues, inspired by the findings of Geirhos et al. [GRM+18]. Others can perturb the image differently for separated spatial frequencies, inspired by the work by Yin et al. [YLS+19]. Lastly, interactive adversarial attacks let the user explore the model's adversarial robustness. Nine of the parameter

Figure 5.9: Top row: original, background blurred, background desaturated, texture influence turned down, lighting influence turned down with texture influence turned down additionally. Bottom row: blurred texture, reduced alpha, reduced saturation, increased contrast (custom GLSL), low frequency band blurred (frequency decomposition).

effects are depicted in Figure 5.9.

We explain each of the parameters below. We use the exact spelling from the interface which uses title case and is sometimes abbreviated. In case of abbreviations, we write the full name next to the interface spelling. Generally, the parameters can be roughly split into pre- and post-rasterization parameters. The pre-rasterization parameters affect the scene before the rasterization stage, the post-rasterization parameters act on the rasterized 2d image.

With this distinction in mind, the **pre-rasterization parameters** are grouped into:

- **"Scene":** General parameters that affect lighting and textures in the scene.
- **"Catness" / "Race Car-ness":** Parameters for morphing the foreground object between dog and cat / fire truck and race car.

The **post-rasterization parameters** are grouped into:

- **"Post Processing":** Simple, single-parameter post processing effects.
- **"Frequency Decomp."** (decomposition): A slightly more complex post processing effect with three parameters.
- **"Adversarial Attack":** Parameters for controlling an adversarial attack on the input image. The computed perturbation image is composed additively on top of the final image resulting from all the other effects.

In the following paragraphs, we describe each of the parameter groups in more detail:

Scene

In the "Scene" parameter group, there are parameters that control textures and lighting in the scene. Providing disentangled controls for texture and lighting lets the user test the respective influence on the activations. While lighting alone primarily induces low-frequency shape cues, textures contain most of the high-frequency information of a scene. The "Scene" parameters are designed as follows:

- With **"Background Blur"**, the user can investigate how the background texture influences the activations. The blur has a wide range, with the background becoming a solid color at the highest level. Blurring the background provides more fine-grained control for testing the model's reliance on background information than swapping the background entirely.
- **"Background Saturation"** complements "Background Blur" for investigating the model's reliance on the background.
- **"Texture Influence"** controls the strength of the foreground object's texture. By default, it is set to 1.0, meaning that the texture is applied normally. Reducing this parameter towards its minimum value of 0.0 interpolates the texture with solid white, diminishing any texture cues on the foreground object.
- **"Texture Blur"** is another parameter that lets the user test the dependence on texture features. Increasing it progressively blurs the texture of the foreground object, with the maximum value leading to a solid color. Reducing texture influence or increasing texture blur can reveal how much a model relies on texture for its decisions. In the results section (Section 7.1), we show how a standard trained model confuses a de-textured cat with various smooth surfaced objects, while the adversarially and Stylized ImageNet trained models still mostly detect pointy-eared animals.
- **"Lighting Influence"** controls the shading of the foreground object. It defaults to 1.0, where the three lights in our scene have full influence on the Blinn-shading of the foreground object. Reduced to 0.0, the object is shaded uniformly with a value of 1.0. This parameter can be used for example to create a silhouette-like shading of the foreground object, removing all features within the object's bounds except the color.
- Finally, the last scene parameter, **"Feat. Vis."** (feature visualization), can be used to investigate feature visualizations as maximally activating images themselves. The control is a checkbox combined with a text field, enabling the replacement of the foreground object with a planar feature visualization texture of the user's choice. The plane with the feature visualization texture still is in 3d space and therefore can be viewed through different camera angles and transforms. On the right of the checkbox, the user can select the feature visualization to be displayed by typing

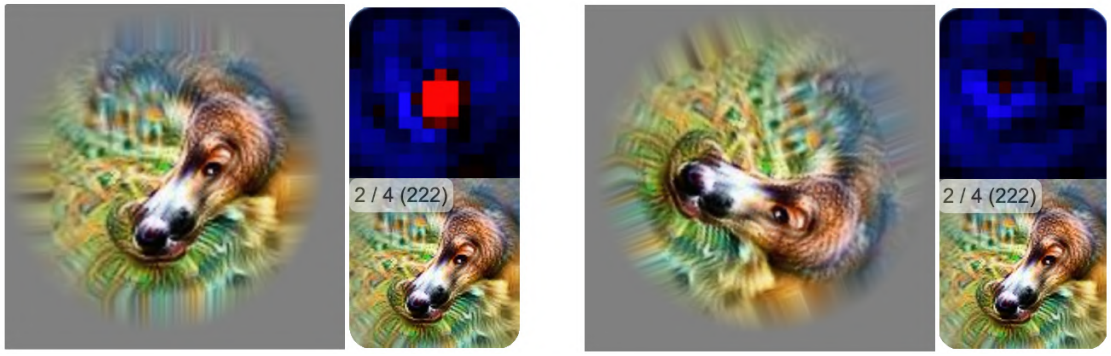


Figure 5.10: Perturber's functionality to display feature visualizations on a plane in 3d space allows investigating how invariant their respective units are to geometric transformations of the input image. In this example, unit 222 from layer mixed4a is tested. Each group shows the input image (left) next to the tested unit's activation map (top right) and feature visualization (bottom right). In the left group, we can see that the feature visualization input without additional transformation causes strong activation in the center of the activation map. In the right group, we can see that after a 60 degree rotation, the strong activation in the center has been completely diminished.

into a text field with the format <layer>:<neuron#>. As feature visualizations maximally activate a respective neuron, having the option to display them in the 3d scene under different camera perspectives and with the additional post-processing options, the user can investigate the invariance of the feature visualization image to those perturbations. An example is shown in Figure 5.10.

Catness / Race Car-ness

As already described in Section 5.2.1, we give the user the possibility to explore the model's response to shape- and texture changes of the foreground object by providing a parameter that seamlessly transforms the dog into a cat, or a fire truck into a race car. The dog / cat class combination is particularly well suited on a geometry level because of similar model topology and proportions, whereas the fire truck / race car combination required more significant distortions. Notably, the rear "water tank" part of the fire truck morphs into the much smaller rear spoiler of the race car. The 3d models within each object pair share their topology, allowing for direct linear interpolation of vertex positions. We also created textures with shared UV layout for each pair. All together, this lets us blend the geometries as well as the textures smoothly between the respective 3d models, resulting in an effective morphing effect (Figure 5.11).

In the "Catness" / "Race Car-ness" parameter group, there are three sliders that control the morphing of the foreground object. As there is an ongoing debate in deep learning research about the ability of CNNs to learn high-level shape features versus just focusing on texture features, we provide parameters to independently control the shape- and

texture mix between dog and cat / fire truck and race car. For instance, the user can observe the network output while gradually changing the dog texture to the cat texture and keeping the dog shape fixed. Alternatively, they could do the inverse and keep the dog texture fixed while gradually changing the shape from dog to cat. The respective parameters are called **"Overall Catness" / "Overall Race Car-ness"** for linked control of shape and texture and then **"Shape Catness" / "Shape Race Car-ness"** and **"Texture Catness" / "Texture Race Car-ness"** for individual control.

Post Processing

This group contains simple post-processing effects that can be controlled by a single parameter. They receive the rasterized 3d scene as input image and are applied successively while preserving the original image dimensions. The post-processing effects do not depend on the geometry of the 3d scene, they could operate on any rgb image. Post-processing effects affecting contrast, brightness, hue and saturation are commonly used in data-augmentation pipelines [SLJ+15, HZRS15]. Providing these operations in the Perturber application lets the user investigate how they affected the model's robustness.

- **"Alpha"** defaults to 1.0 and simply darkens the image towards black when reduced.
- **"Hue"** goes from -0.5 to 0.5, which corresponds to -180/180 degrees on the color wheel. This parameter shifts each pixel's hue value accordingly.
- **"Saturation"** defaults to 1.0, reducing the parameter towards 0.0 de-saturates the image until the result is a purely grayscale image at 0.0.
- In the **"Custom Parameter"**, the user finds a text field where they can write their own GLSL code, taking a single parameter which is controlled by the slider. The code snippet defaults to code for contrast adjustment.
- **"Patch Shuffling"** is an effect inspired by Zhang and Zhu 2019 [ZZ19] and can reveal a model's sensitivity to global structure, which gets highly disturbed by patch shuffling. The image is divided into a grid of k by k cells, which are then randomly re-ordered. A k value of 1 leaves the image unchanged. Examples analagous to our implementation are depicted in Figure 5.6 (e) and (f), with k values of 2 and 4 respectively.

Frequency Decomposition

The frequency decomposition, labelled "Frequency Decomp." in the interface, is a three-parameter effect that splits the image into two separate frequency bands (low/high). This is done by Laplacian decomposition: We blur the original image, which is our low-frequency band. We subsequently subtract this low-frequency band from the original image and thereby get the high-frequency band. Summing them together results in

the original image. The user can investigate phenomena such as the one described by Yin et al. 2019 [YLS+19], where the authors show that adversarially trained networks are less sensitive to high-frequency perturbations but more sensitive to low-frequency perturbations compared to standard trained models (Their results are shown in Figure 5.5).

The **"Cutoff"** parameter controls the separation frequency (blur radius of low frequency image). Having the image separated into bands lets us manipulate them individually. For both frequency band images we then provide three parameters each:

- **"Low Alpha" / "High Alpha"** control the magnitude of the respective frequency band image. A fully reduced low frequency alpha replaces the low frequency content with uniform gray. A fully reduced high frequency alpha just leaves the low frequency content, being the blurred image with blur radius according to the "Cutoff" frequency. Images with reduced high and low alpha can be seen in Figure 5.12.
- **"Low Sigma" / "High Sigma"** control the blur of the low and high frequency images separately. Examples with blurred low and high frequency image can be seen in Figure 5.12.
- **"Low Hue" / "High Hue"** let the user change the hue of the low and high frequency image separately. This parameter is not inspired by existing work, but can be used along alpha and blur to test a model's sensitivity to perturbations in a specific frequency range.

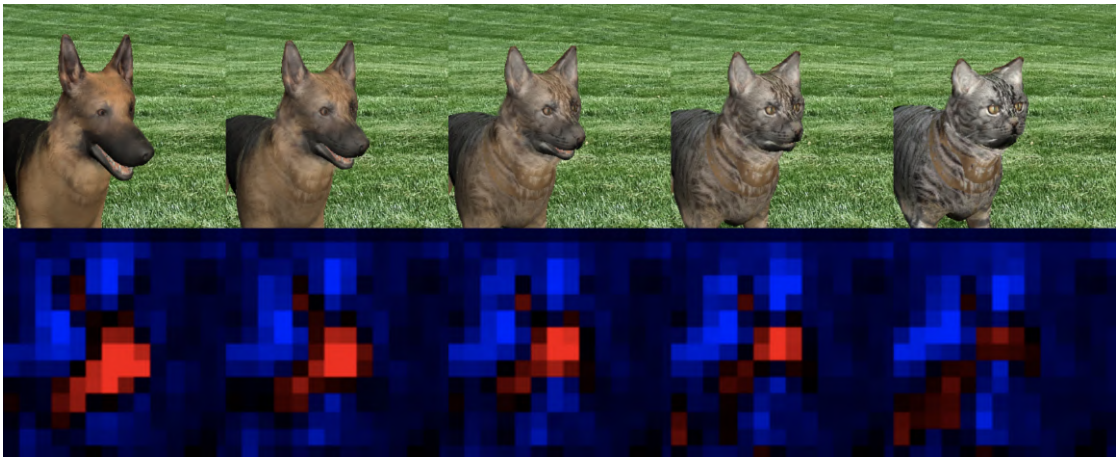


Figure 5.11: The top row shows various steps of the interpolation from dog to cat. Bottom row shows the response of a right-facing-dog detector of Layer "mixed4a" to above shown images. A gradually vanishing response can be observed.

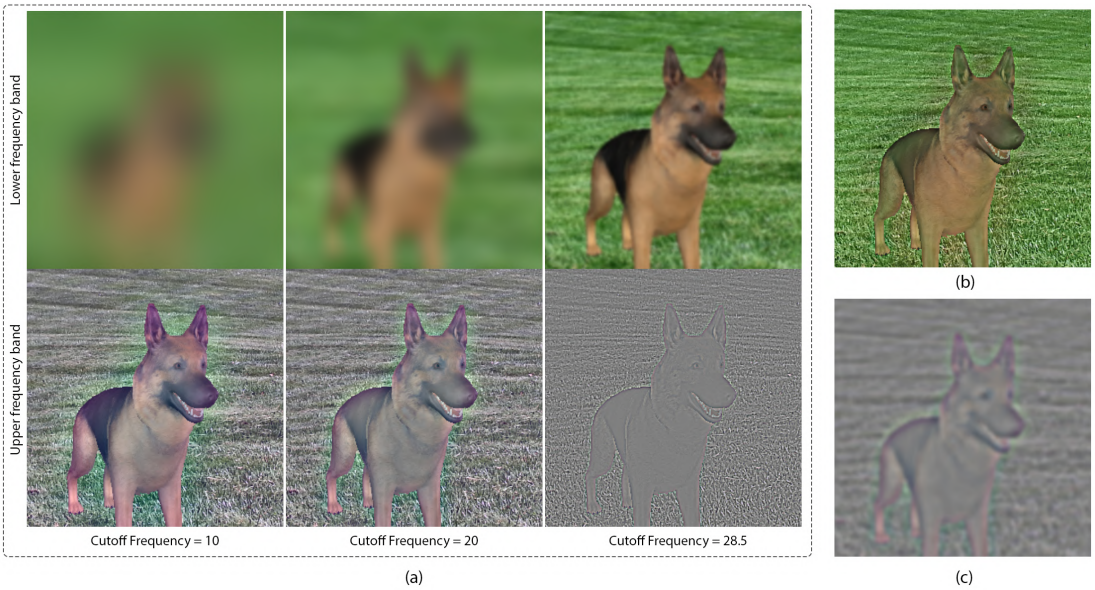


Figure 5.12: The three columns in (a) show the low and high frequency image for increasing cutoff frequency settings. Low and high frequency images with high and low alpha parameters set to zero respectively are shown in the top and bottom row. The right-most column shows the effect of blurring the low (b) and high (c) frequency band only. For (c), the low frequency alpha has been reduced for better visibility.

Adversarial Attacks

The manipulation capabilities described so far have been designed to test a model's robustness with perturbations that are clearly visible to humans. We are not interested in limiting these parameters to preserve similarity to the original under a certain metric. Conversely, adversarial attacks can perturb the image in a way that is often hardly visible to humans, while changing the CNN's output significantly. Understanding the difference between adversarial training and other robust training methods, such as training on Stylized ImageNet is one of Perturbers goals, inspecting the models' reaction to adversarial attacks is a key feature of the application. We give the user the ability to perform PGD adversarial attacks (explained in Section 2.2.1) on the current scene image. The user can choose:

- the model to generate the attack from (Model 1, Model 2, Edited Model),
- the target class, or simply to suppress the original prediction (untargeted attack), and
- and the attack epsilon and L_p-norm (L_2, L_∞).

Pressing the "PGD Step" button will perform one PGD step. The step magnitude is one-eighth of the chosen epsilon value. The first step takes a comparatively long time (around 10-20 seconds), as the gradient function needs to be computed. The gradient function is then cached for subsequent calls. Letting the user press a button for each individual PGD step was a design decision we made. It was based on the hypothesis that having to wait for rather slow iterative updates would be a much poorer user experience than controlling each step manually, thus being able to inspect activation changes after each step. The top row in Figure 5.13 shows the results of strongly visible adversarial attacks for various models

Once the adversarial attack is initiated, the resulting image (original image + perturbation vector) is overlaid as a static texture on top of the scene. As the adversarial attack is an iterative process, scene manipulation after the first iteration is not possible. We also show class probability output for the original top class before the attack (top left), after the attack (top right), as well as class probability for the current top class (bottom right) as text overlay. Clicking into the scene area after an attack has been initiated resets the attack and gives the user the ability to manipulate the scene with other parameters once again.

With an active attack overlay, the user can use the "Attack Alpha" to fade the perturbation, as well as the "Original Alpha" to fade the original image to solid 50% gray, leaving just the attack vector when reduced to zero. This allows for a more interactive analysis of the model's behavior to attack strength, as fading the attack can be computed much quicker than calculating it in the first place. Also, the attack vector itself can be inspected by fading out the original image.

5.3 Neuron Activation View

The neuron activation view is the central interface for inspecting the effect on intermediate convolutional layers. For each of the three models (Model 1, Model 2, and Edited Model), activation maps are shown interactively for a selected layer and a selected group of neurons. Feature visualizations are juxtaposed directly below to provide the user with a hint at what the neuron above responds to, but also as an independent explorative element.

Various visualization parameters let the user adjust the feature visualization to the current use case. These parameters can be seen in Figure 5.14, orange box:

- **"Naive Parameterization":** By default, we show feature visualizations with a parameterization that better distributes the gradient onto all spatial image frequencies (spatial decorrelation as explained in Section 2.3.1). For Model 1 and Model 2 and whichever model is assigned to these slots, this means that the feature visualization is parameterized as a frequency spectrum, transformed to an image by inverse Fourier transform before being used as input for the network. For the Edited

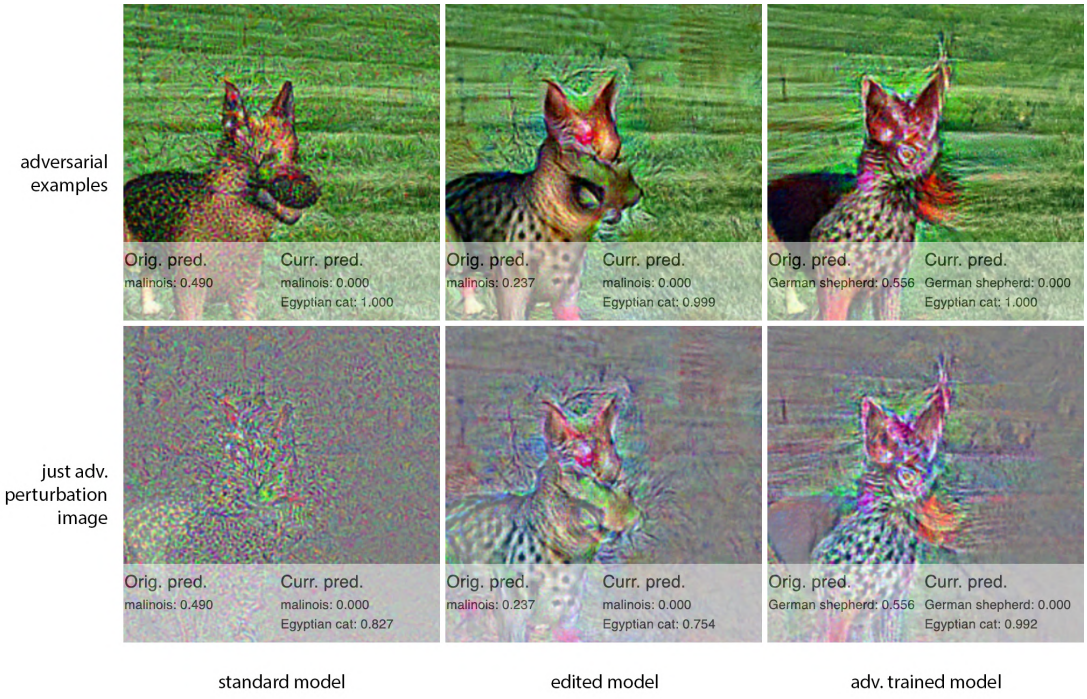


Figure 5.13: Top row: Strong ($\epsilon = 100.0$) adversarial examples for the standard model (left), the edited model with early layers until "mixed3a" from the adversarially trained model, others from the standard model (middle), and adversarially trained model (right). Bottom row: respective isolated perturbation vectors.

Model, a Laplacian pyramid [BA87] parameterization is used, having a similar effect as the Fourier parameterization. When "Naive Parameterization" is checked, the feature visualizations switch to a version that is parameterized in pixel space. This leads to a much stronger visual difference between feature visualizations for standard trained and adversarially trained models. The use of the Fourier parameterization inductively biases the result towards a more equalized distribution, therefore we give the user the option to choose. Also, "Naive Parameterization" feature visualizations do not use decorrelated color space (see Olah et al. [OMS17], Section "The enemy of feature visualization"). The difference is shown in Figure 5.15.

- **"Show Activations":** Un-ticking this option will hide the interactive activation maps. This saves computational resources and can be useful if the user is more focused on inspecting feature visualizations alone than on investigating the model's response to input changes.
- **"# of Neurons":** As a compromise between showing an extensive overview and providing focus and computational speed, visualizations for four neurons are shown by default. The user can choose to show up to 8 neurons when a larger

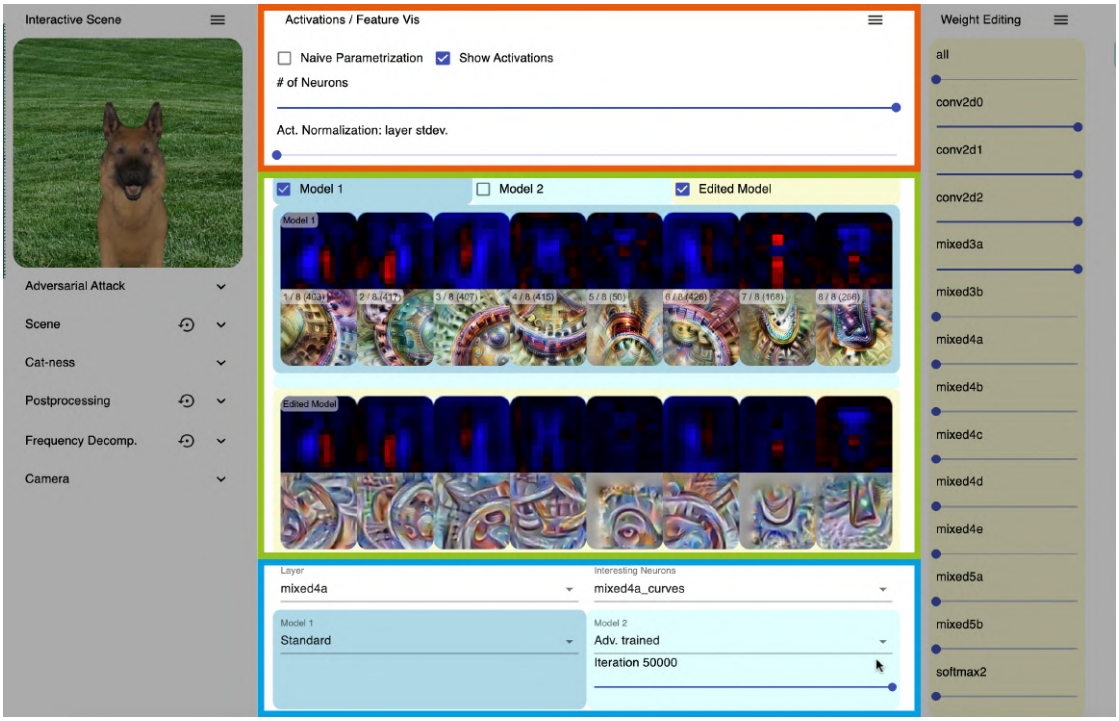


Figure 5.14: This figure shows the Neuron View, located between the Interactive Scene on the left and the Weight Editing pane on the right. It contains visualization parameters (orange box), visualizations (green box), and model parameters (blue box). In the shown configuration, number (#) of Neurons is maxed out at 8, which makes it possible to display all neurons of the selected group ("mixed4a_curves") at once. "Naive Parameterization" is ticked off, so feature visualizations computed with Fourier / Laplacian pyramid parameterization are shown. "Show Activations" is checked, so activation maps are shown for each model. "Model 2" is ticked off, saving GUI space and computation time. For the "Edited Model", feature visualizations have been generated interactively by pressing the "Play button" visible in the center of the feature visualization row.

overview is important, or reduce their number when focusing on a smaller set or an individual neuron. Reducing the displayed number of neurons can also be beneficial when the efficient use of limited computational resources is important, for instance when interactively generating feature visualizations for the Edited Model (further explained in Section 5.4.1).

- **"Act. Normalization"** (activation normalization): To display activation values, we map positive values to red and negative values to blue. In order to provide a visualization with well-distributed brightness values, the activation values have to be normalized such that the values with the highest magnitude map to one. By default, the activations are normalized by division with four standard deviations of the whole layer's activations. Presented in this way, the shown activations can

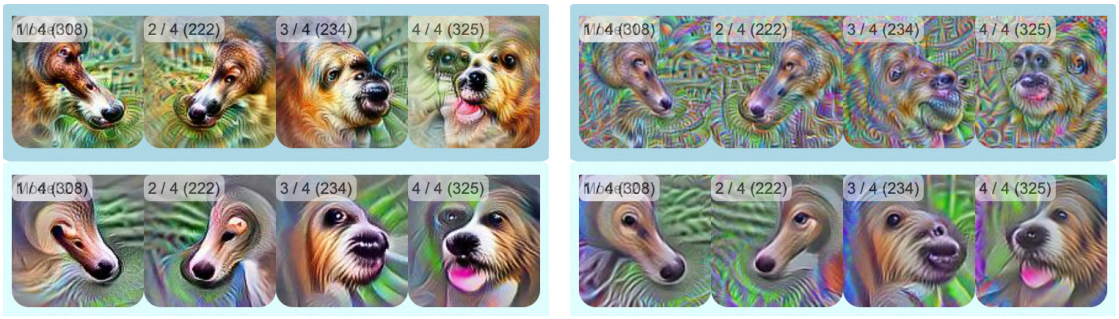


Figure 5.15: Feature visualizations for various neurons from layer mixed4a. On the left side, they have been generated with spatial and color decorrelation, whereas on the right side, they use "naive parameterization". The top row shows feature visualizations for the standard trained model, where a strong difference in visual appearance can be seen between the parameterizations. The bottom row shows feature visualizations for the adversarially trained model, where the naive parameterization mostly impacts the colors, but not the spatial structure.

only be seen in relation to other activation values of their layer. When the absolute activation values are of interest, or when the selected neurons have activations in an outlier value range, a manual adjustment of the normalization divisor can be useful. The Activation Normalization slider lets the user gradually adjust the normalizing divisor in an exponential way. The slider value is the log_{10} of the divisor if it is larger than zero. A slider value of zero enables standard deviation-based normalization. Figure 5.16 shows an example where manually setting the normalization divisor is useful.

conv2d0	3
conv2d1	8
conv2d2	14
mixed3a	18
mixed3b	21
mixed4a	20
mixed4b	8+4
mixed4c	5+4
mixed4d	3+4
mixed4e	0+4
mixed5a	0+4
mixed5b	0+4

Table 5.1: Number of neuron groups per layer. "+4" represents the cat, dog, fire truck, and race car relevant neurons we identified ourselves, the other numbers represent the number of neuron groups identified within the Circuits project.

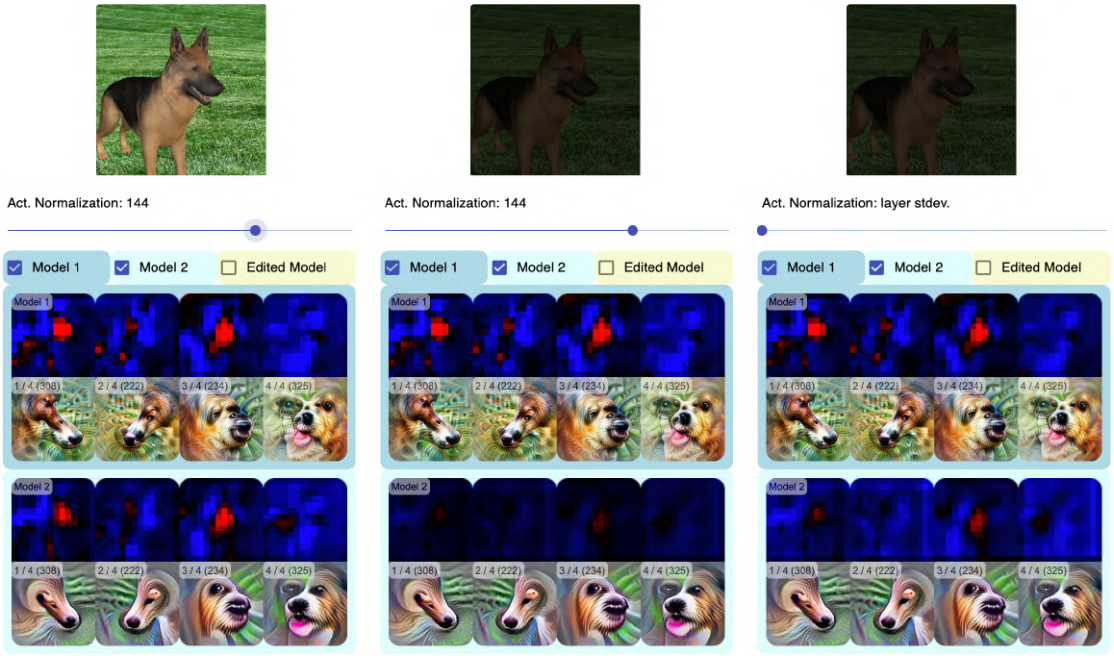


Figure 5.16: An inspection scenario where manual activation normalization is useful: On the left, the neuron view shows the activations for the unaltered input image, with activation divisor set to 144. In the center, we can see how the activations change after the input image has been darkened by reducing the "Alpha" parameter. The adversarially trained model's activations in the shown channel have clearly decreased in magnitude (Model 2), whereas the standard trained model's activations are hardly affected (Model 1). On the right we can see the same visualization using the default normalization, based on layer-wide standard deviation: The different behaviour between the two compared models is mostly lost in the visualization.

Beneath the visualization pane, the parameters for controlling model- and neuron selection are located (Figure 5.14 blue box).

To allow the quick selection of interesting neuron groups, we provide pre-selected neuron groups and categories for each layer, mostly taken from the Circuits project [CCG+20]. The top row contains the shared layer- and neuron group parameters. The user can select a layer of interest on the left, for which the right selection box is then populated with interesting neuron groups/categories. Examples of these neuron groups can be seen in Figure 5.17. The number of neuron categories varies per layer. Table 5.1 shows these numbers. These numbers reflect the fact that earlier layers' features lack the expressive power for a large variety of neuron categories. Layers in the midsection of the network ("mixed3b", "mixed4a") have a large number of identified categories, later layers seem to be increasingly polysemantic which makes identifying neurons in a distinguished way harder. This is reflected again in their low number of neuron categories, although this number might rise with the progress of the Circuits project. We provide our own selection

Figure 5.17: Examples of neuron groups from different layers (from the Circuits project [CCG+20]).

of cat, dog, fire truck and race car relevant neurons for the later layers, which we identified by using *Summit* [HPRPC20]. Summit displays neurons which are important for the prediction of a selected ImageNet class for each layer of Inception V1. We chose several highlighted neurons per layer for "fire truck" and "race car". For "dog" and "cat", we aggregated important neurons from multiple dog and cat breeds.

The model selection parameters below are color-coded with light blue for Model 1 and a slightly darker blue for Model 2. Model 1 and 2 are slots that can be arbitrarily assigned. Each slot has a menu for the training dataset with options "Standard", "Stylized" and "Adv. Trained". Below is a slider that can be used to select a checkpoint at selected iterations during fine-tuning. Iteration 1 of "Adv. Trained" is identical to the "Standard" model, which does not allow choosing the iteration. The iteration slider loads the respective checkpoint only upon releasing the mouse. This allows for efficient exploration of feature visualizations along training checkpoints, as loading the model is a lengthy process taking at least a few seconds and potentially much more depending on the internet connection speed of the user.

The color-coding for the model slots is used anywhere in the application where model-specific information is displayed or selected. In addition to the two shades of blue for Model 1 and Model 2, a yellow color is used for the edited model. We describe the editing model and the corresponding editing interface component in Section 5.4.

The central part of the neuron view is the activation and feature visualization view (green box in Figure 5.14). It consists of one row of activation maps and one row of corresponding feature visualizations *per model*. To avoid cluttering the interface with visualizations that are not of current interest, the user can enable / disable the visualization pair for each model with a checkbox. Often the user might want to focus on inspecting one model individually or comparing two models with each other instead of looking at all three models. The checkboxes share the common color encoding for Model 1, Model 2 and

Edited Model. The visualization rows for each model are also framed with the respective color. This leads to an intuitive association of the visualization row with the model. The visualization rows are additionally labelled in the upper left corner.

5.4 Weight Editing

The interactivity of Perturber allows for direct inspection of model activation changes in response to input scene changes. As the model activations depend on a) the input to, and b) the parameters of the model, a natural extension of our interactive model exploration is the interactive editing of the parameters (meaning the weights) of the model.

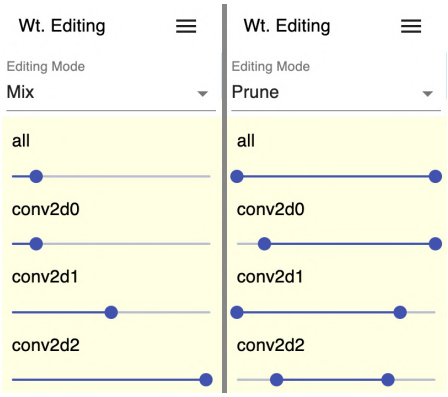


Figure 5.18: Top part of the weight editing interface for mixing (left) and pruning (right). The editing mode is chosen in a drop-down menu ("Editing Mode") at the top of the component. The "Prune" mode displays range sliders which allows the user to define a relative range from minimum- to maximum-magnitude weights of the respective layer to be kept, while out-of-range weights get multiplied by zero. Sliders continue below "conv2d2" for all other layers.

While many ways of editing weights could be imagined, ranging from scaling and shifting to hand crafting weights [CGC+20], we restrict Perturber to two options, whose interface components can be seen in Figure 5.18:

- **Mixing weights from two different models.** We give the user the possibility to mix weights from Model 1 and Model 2 together. Mixing weights from two models with similar feature representations lets the user explore how mixing the weights influences the prediction result, as well as robustness. As our provided models are either a base model or one of its fine-tuned variants, feature alignment is relatively strong. We found that combining layers from two feature-aligned models does not necessarily destroy the prediction result and we provide this functionality in our application to be further explored. There is one slider for each of the "conv2d" layers and for each "mixed" block. This slider controls the interpolation between the weights from Model 1 and the weights from Model 2 for the respective layer or

block. In high level terms, a layer uses the weights from Model 1 when its slider is set to 0 (left side) and the weights from Model 2 when it is set to 1 (right side). In between values linearly interpolate between Model 1 weights and Model 2 weights. Formally this can be written as

$$\mathbf{W}_l^{mixed} = \mathbf{W}_l^{m1} \cdot (1 - \alpha_l) + \mathbf{W}_l^{m2} \cdot \alpha_l \tag{5.1}$$

where $\mathbf{W}^{mixed}$ are the resulting weights for the edited model, $\mathbf{W}^{mk}$ are the weights of Model k, α is the slider value and l the layer/block index. For the blocks, all included layers share one interpolation parameter.

- **Pruning weights based on magnitude.** Another weight editing method we provide is pruning by magnitude. Pruning is often used for model compression when limited memory is a concern, such as on mobile devices. We implement pruning to give the user the possibility to investigate network activation and feature visualization changes in response to switching off entire branches within the network. There exists a multitude of pruning techniques, most of which are based on either weight or gradient magnitude. Our pruning method is a simple variant where entire filters are cancelled out by their average absolute input weight magnitude.

  We calculate the average absolute value over the spatial- and input filter dimensions for each weight kernel. We get a vector $\mathbf{m}$ with a length of the number of output channels. For a convolutional layer this can formally be written as

$$\mathbf{m} = \sum_y \sum_x \sum_i |W_{yxio}| \tag{5.2}$$

  where y and x are indices for the spatial dimensions of the 4 dimensional weight kernel $\mathbf{W}$, i is the input channel index and o is the output channel index. For a fully connected layer this would be simplified to

$$\mathbf{m} = \sum_i |W_{io}| \tag{5.3}$$

  We then sort the values of vector $\mathbf{m}$ in ascending order:

$$\mathbf{s} = argsort(\mathbf{m}) \tag{5.4}$$

  where $\mathbf{s}$ contains the sorted indices of averaged weight magnitudes. From $\mathbf{s}$ we get to the pruning mask vector $\mathbf{p}$ by setting the value of $\mathbf{p}$ to 1 if its index within $\mathbf{s}$ is in the kept range, and to 0 otherwise:

$$p_{s_n} = \begin{cases} 1, & \text{if } r^l \cdot O < n < r^u \cdot O \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

  where r^l and r^u are the lower and upper pruning range values, O is the number of output channels and n indexes $\mathbf{s}$. In other words, elements of $\mathbf{p}$ get assigned 1

if the average weight magnitude of the corresponding output channel is within a percentile range delimited by r^l and r^u.

Finally, we multiply the original weight kernel $\mathbf{W}^{orig}$ by the pruning mask vector $\mathbf{p}$ to get the pruned kernel $\mathbf{W}^{pruned}$:

$$\mathbf{W}^{pruned} = \mathbf{W}^{orig} \cdot \mathbf{p} \tag{5.6}$$

5.4.1 Interactive Feature Visualizations

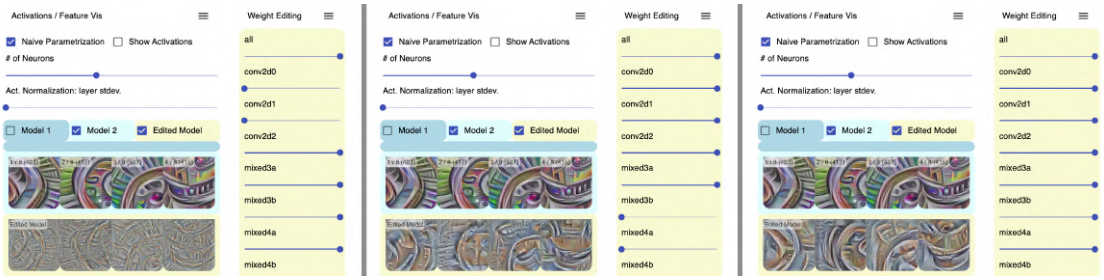


Figure 5.19: Using the interactive feature visualization component, the user can explore the effects of weight editing onto the feature representation. In the three screenshots, different weight mixing configurations are depicted, along with their different interactive feature visualizations (bottom row). The example is explained in detail in the main text.

To let the user compare feature visualizations between the base models and the Edited Model, we provide a mechanism to generate feature visualizations on the fly for the Edited Model. As explained in Section 4.1.2, feature visualizations for standard and adversarially trained models have highly different appearance. We therefore assume that generating and inspecting feature visualizations while editing a model's weights can be informative about the adversarial robustness of the Edited Model's current state. We present a quantitative experiment supporting this assumption in Section 7.2.

Figure 5.19 shows an example how the interactive feature visualizations can help exploring the effects of weight editing onto the feature representation. In the left screenshot, the Edited Model uses weights from the standard trained model (assigned to Model 1) for the first two layers and weights from the adversarially trained model (Model 2) for the rest. The feature visualizations have been generated for several curve detectors from layer "mixed4a". The top row shows feature visualizations for Model 2, the bottom row shows feature visualizations for the Edited Model. We can see that the bottom row is much more high-frequency heavy than the top row. In the central screenshot, the last two layers with influence on the shown feature visualizations ("mixed3b", "mixed4a") use weights from the standard trained model and the rest use weights from the adversarially trained model. Looking at the feature visualizations generated for the Edited Model (bottom row), we can see that in the center screenshot they contain more low-frequency structure than in the left screenshot. We conjecture that the first two layers are more

important for the low-frequency structure of the adversarially trained model's feature representation than the later layers. In the right screenshot, the Edited Model only uses weights from the adversarially trained model. The generated feature visualizations, as expected, look almost identical in structure to the pre-computed ones for Model 2. The remaining visual difference can be explained by the slight implementation differences mentioned in Section 6.2.3.

Figure 5.20: The top row shows feature visualizations for a pair of fur-detecting neurons from Layer "mixed4a" created with python-based feature visualization library Lucid, the bottom row shows feature visualizations for the same neurons generated by our TensorFlow.js implementation, adopted from LucidPlayground [SW19]. Left, two neurons are from a standard trained model, right two neurons are from corresponding adversarially fine-tuned model.

5.5 Prediction View

The Prediction View (5.21) shows the top 5 classification results for each model. This allows the user to observe the classification changes resulting from input changes or model editing operations, and to compare those between models.

The top result is represented by a large class example, as not all ImageNet classes are easy to understand from just their name. This also provides an easy to grasp visual representation, leading to a more intuitive exploration. Hovering over a different top 5 class than the first one replaces the example image with an example from that respective class.

As running the full model on the input image is a rather time consuming operation, we provide toggles to enable and disable this component for each model individually. The

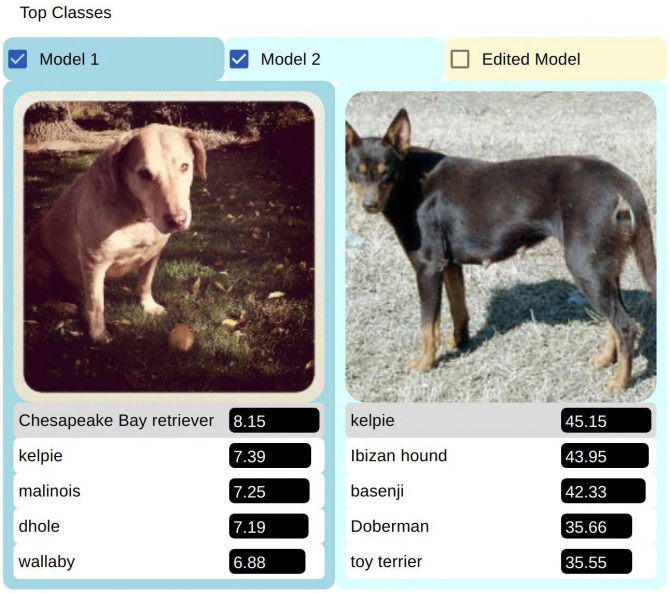


Figure 5.21: The Prediction View displays the Top 5 classification results for each model. The top result is represented by a large class example. Below the class example there are five rows with the class names and bars representing respective logit values of their class. The bar length is normalized by the maximum value. We also provide absolute values inside the bars as written numbers to represent absolute scale.

Prediction View is the most significant bottleneck of the whole application, disabling it when not needed can lead to a massive speed up when computational power is limited.

CHAPTER 6

Implementation

This chapter describes implementation details for both the preliminary experiments and the visual analytics application. Our early feature visualization experiments were first performed independently from the development of the Perturber visual analytics application. The experiments used a different eco-system based on Python, whereas the visual analytics application is based on JavaScript. Bridging the gap between them and making training checkpoints generated during our preliminary experiments available in our interactive application was not trivial and could only be achieved by significant workarounds.

We will describe the implementation details for our preliminary experiments as well as the model conversion workarounds in Section 6.1. The visual analytics application implementation details will be described in Section 6.2.

6.1 Preliminary Experiments

In this section we provide implementation details for the various components of the data generation pipeline. The data generation pipeline consists of model training, generation of feature visualization, and model conversion.

6.1.1 Adversarial Training

Our training code is based on code by [TSE+19], which uses Tensorpack [W+16] for multi-GPU training. For our preliminary experiments with ResNet 18 we modified the code to facilitate fixed initialization and a deterministic dataset schedule. This was done by disabling non-deterministic parallel data pre-processing from Tensorpack. For generating the weight kernel visualizations in Figure 4.1, we wrote a custom callback function that attaches the reshaped and normalized kernel as an RGB image to Tensorpack's training-"monitors" via the *put_image* function.

We restrict ourselves to L_2-bounded adversarial training. We experimented with multiple ϵ values ranging from 0.0001 to 1.0, where $\epsilon = 1.0$ denotes the length of the full range of floating point intensity values, corresponding to a value of 255 for images of data type *uint8*. The significant results from these training runs have already been discussed in Section 4.1.

The training code requires Tensorpack version 0.8.5. For freezing the model weights and exporting the trained model to a TensorFlow *Protobuf* file, we use Tensorpack's "ModelExporter" class, which requires Version 0.9.8.

6.1.2 Feature Visualization

For the generation of our feature visualizations we use the Lucid feature visualization library at Commit 246ef62. We modified the code to facilitate correctly fixing the random initialization of the visualizations by seeding NumPy's random generator in the *render* function of the *optvis* module in addition to the already available fixed seeding of TensorFlow's random generator.

We mostly use *Lucid's* default parameters for generating feature visualizations and vary only layer and neuron in the "neuron" objective. For our non-Fourier-parameterized feature visualizations we pass on *fft=False* as well as *decorrelate=False* to the *image* function of the *param* module.

6.1.3 Conversion to TensorFlow.js

Perturber is built using TensorFlow.js (TFJS) and therefore requires models to be provided in the custom TFJS format, which consists of a JSON file containing the graph definition as well as binary files containing the weights. TFJS comes with a converter that can convert a multitude of TensorFlow model files into TFJS-compatible model files. The converter needs to receive a *Keras* Layer Model file as input.

We therefore re-implemented our models as Keras Layer Models and initialized them with the weights of the Protobuf model files exported from Tensorpack. We then saved *H5* files with Keras's *Model.save* function before converting them with the TFJS model conversion tool.

Inception V1, our model architecture, contains local response normalization (LRN) layers. This type of layer is neither available in current versions of Keras nor in TensorFlow.js. We adopted an LRN layer implementation for our Inception V1 Keras re-implementation. Besides creating the respective entry in the Keras model file used for conversion to the TFJS format, the working LRN implementation in Keras allowed us to test our Keras model against the original Inception V1 model in native TensorFlow, and verify that they output identical activations in various layers.

While a custom LRN layer implementation leads to a respective entry in the Keras model file, as well as the JSON file resulting from the TFJS conversion, a corresponding implementation has to be available on the loading side as well. On the TensorFlow.js side

we therefore also had to make some modifications to get the LRN layer to work correctly. We describe this further in the next section.

6.2 Perturber

Our final application has evolved from a minimal proof-of-concept prototype to a feature-rich but complex application and then to a more clear and concise reduced version that balances user experience with amount of features. In the following subsections we will provide some background about the decisions regarding used frameworks and libraries.

6.2.1 React.js as GUI Library

React.js serves as the main GUI library and had strong influence on our project structure. It induces a top-down hierarchy of components which either contain their own state or receive and modify state from parent components. The decision to use React was made because of the following considerations:

- React.js is a mature GUI library with widespread adoption among web developers.
- We had successfully used React.js in other projects and we were familiar with its capabilities and restrictions.
- React-three-fiber is a project that integrates the Three.js 3d rendering library into the React component model and provides a React development experience without restricting functionality.

The decision for React.js against alternative GUI libraries was therefore mostly based on prior personal experience, but solidified by the perfect integration of Three.js. We did not evaluate alternatives because we did not find it necessary.

In React, the typical way to share state between leaf components is to store it in the lowest shared parent, causing the re-rendering of the entire sub-hierararchy on state change. We therefore use *zustand* (https://github.com/pmndrs/zustand) for state management where appropriate. This allows us to efficiently pass changed state in a publish-subscribe way between components in different branches far down the component hierarchy without causing the re-rendering of all parent components. This does not apply to the main rendering loop. Any change in the input parameters results in the following rendering sequence being executed (as depicted in Figure 6.1):

1. After the Scene Renderer has rendered a new frame, it passes the data to the Model for inference.
2. The Scene Renderer receives back the activation data.
3. The Scene Renderer updates the Root's state by calling *setActivationData(newActivationData)*.

4. The Root got its state modified and has to re-render itself and all childs.

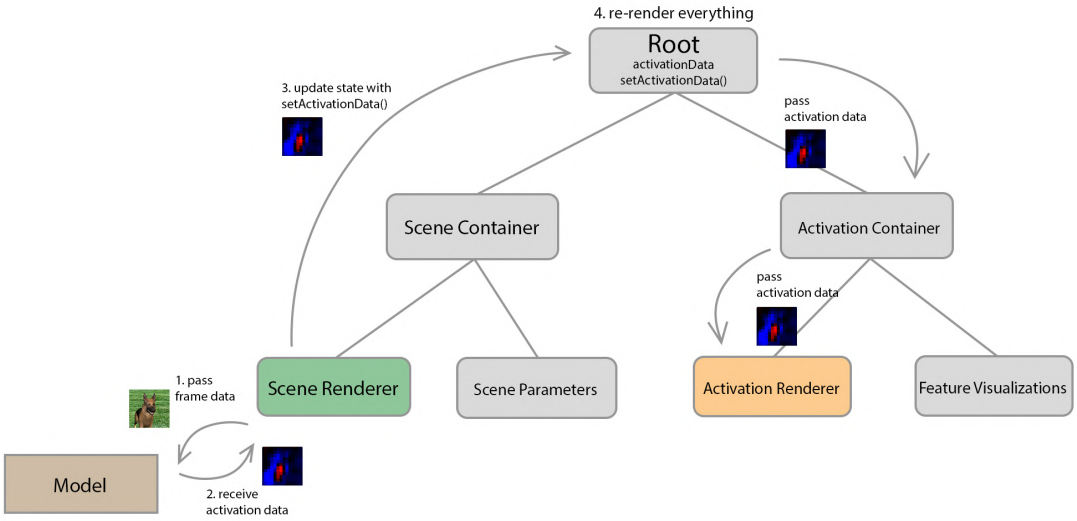


Figure 6.1: Example of a typical way to pass data from one child component to another child component in a different branch: The Scene Renderer component is in a different branch than the Activation Renderer, their shared data (activationData) is stored in the nearest root.

6.2.2 Material-UI

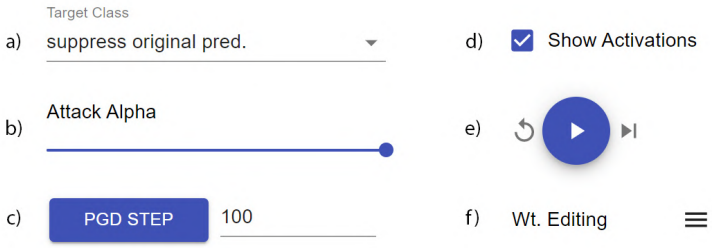


Figure 6.2: Examples of Material-UI components that we used: a) Select, b) Slider; c) Button and Textfield; d) Checkbox; e) IconButtons (white) and Fab (blue); f) ListItem used as header, with menu icon to symbolize collapsible menu.

Material-UI contains many pre-made customizable React components based on Material Design. Using a component library such as Material-UI leads to a significant development speed-up while keeping a consistent design. Examples of Material-UI components used in our application are shown in Figure 6.2.

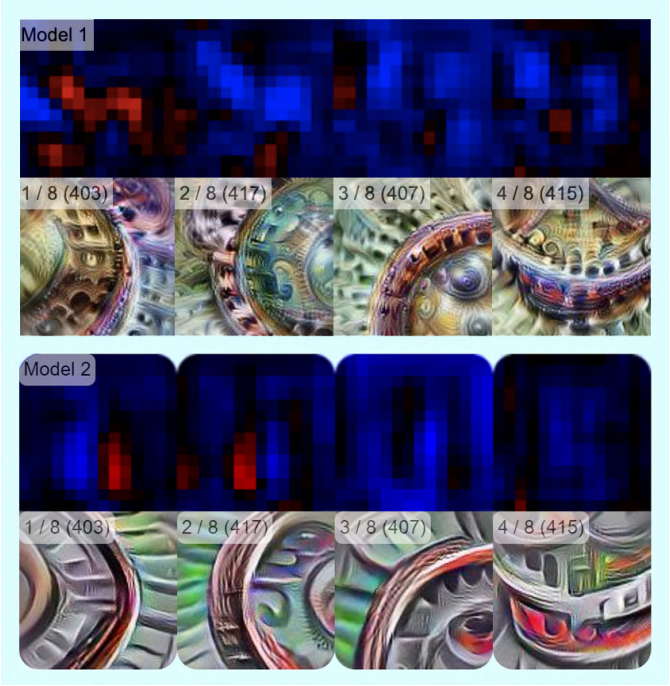


Figure 6.3

The rounded corner style found in Material-UI leads to a more pleasing, "softer" visual appearance. We adapt the remaining interface to follow this style by rounding the corners on most otherwise sharply rectangular shapes, such as parent container boxes.

For the juxtaposition of activation maps to feature visualizations, we additionally use rounded corners for grouping the activation map - feature visualization pair for each neuron. As can be seen in Figure 6.3, without this design decision the visual association of the pairs is much harder and slower to perceive.

6.2.3 TensorFlow.js

TensorFlow.js is a JavaScript machine learning library utilizing the GPU via WebGL for efficient parallel execution of machine learning related operations. In contrast to TensorFlow 1.x, it natively (and exclusively) runs in eager execution mode.

We use TensorFlow.js for all neural network inferencing operations and for gradient ascent in adversarial attacks and feature visualizations. The CNN inferencing operations represent the most time-consuming component of our application. We therefore optimize the data flow to only run inference when necessary. This mostly affects the way we handle the render loop, which constantly renders new frames. When the camera is stationary and no parameters are updated, sending the frame data to the inferencer each time a new frame arrives would be highly inefficient and would waste a lot of computation while idling. We implemented a mechanism that requires each parameter-, camera- or model

change to *explicitly* invalidate the current activation data, triggering the new activation data to be computed for the next frame after the change only.

Another crucial implementation detail is the way the application handles receiving data from the GPU. While executing TensorFlow.js operations, the data is stored in Tensor objects. They are not directly readable by the client (non-GPU) side. To get data from Tensors, one has the choice between asynchronous and synchronous data fetching. Calling *Tensor.data()* returns a JavaScript Promise, which gets fulfilled with the data as function parameter. *Tensor.dataSync()* returns the data directly as a *TypedArray* object.

We experimented with both methods and found the following advantages and disadvantages for each:

- **Synchronous** data fetching blocks the GUI for the whole duration of the inference operation which takes longer than 16 ms (required for smooth 60 Hz playback) in most usage scenarios, sometimes exceeding 200ms and more (full inference with logits for three models). This can lead to a very stuttery experience when changing camera angle or moving parameter sliders. On the other hand, each input scene change gets reflected in the displayed activation visualizations **before the next input scene update**. The chronological order

  input scene change -> inference -> visualization -> next input scene change

  gets therefore preserved, which is important for the user to link the input scene to the activation visualizations.

- **Asynchronous** data fetching frees the GUI from having to wait for the time-consuming inference operations to finish. This leads to a more fluid GUI, always being able to render at a high frame rate and thus providing a smooth experience. The disadvantage is that the components displaying the model data often severely lag behind the current input scene. This makes it hard for the user to judge if the current parameter change is reflected in the activation visualization or if the user still has to wait for a few tenths of a second for the changes to be displayed. In the case of a MacBook Pro (with comparatively slow GPU), asynchronous data fetching even lead to the activation visualization components not updating at all during continuous paramter changes, requiring the user to pause the change to view the updated visualization.

We chose the synchronous data fetching because the overall tradeoff seemed more sensible. A slower user experience that consistently displays activations according to the input image is preferable to a smoother user experience where the user can not be sure if the current activation visualization reflects the input scene. Performance measurements will be presented in Section 7.4.

For generating **interactive feature visualizations** we adopt the code from Lucid Playground [SW19]. It is also based on TensorFlow.js [STA+19], which lacks an inverse

fast Fourier transform (iFFT) operation that does exist in TensorFlow [AAB+15], necessary for distributing the gradient well among all frequency bands. We use a Laplacian pyramid [BA87] parameterization to approximate the behaviour otherwise achieved by iFFT. Switching the naive parameterization toggle changes the parameterization between a single pyramid layer and 5 pyramid layers. Empirically, 5 layers provide a good tradeoff between speed and approximating a full Fourier parameterization.

6.2.4 Three.js

Three.js is a JavaScript library that greatly accelerates the usage of WebGL functionality by providing a more high level programming interface to rendering 3d scenes. We use Three.js wrapped by the React.js renderer "react-three-fiber"(https://github.com/pmndrs/react-three-fiber). This allows us to set up our 3d rendering in a declarative way and relieves us from difficulties that can arise when one wants to integrate the Three.js render-loop into a React component hierarchy. We exclusively use custom GLSL shaders for rendering background and foreground of the 3d scene, allowing us to take full control of the rendering aspects. The foreground object in the 3d scene consists of two meshes that are interpolated on the GPU according to a parameter passed as shader uniform.

For post-processing effects we also use custom GLSL shaders. We use the *Effects Composer* from Three.js for the multi-pass post-processing pipeline and a custom render pass with internal multi-pass rendering for the frequency decomposition post-processing effect.

For "Background Blur" as well as for "Texture Blur", we use the *textureLod* GLSL function to access a higher mip-mapping level (corresponding to lower-resolution versions of the respective texture) on the GPU, which is much more efficient than a kernel-based box- or Gaussian blur. We interpolate between two mip-mapping levels for any non-integer parameter value.

CHAPTER 7

Results

In this chapter we will present the results of our work. We will present an extensive case study with five expert participants as well as discoveries made by us while using Perturber. Then we will show performance measurements.

7.1 Case Study

We conducted, in total, five case studies with expert researchers in deep learning. Four of our five case studies were held as online video conferences where the participants shared their screens during the session. In two cases, there was one researcher guiding the participant while another researcher simultaneously transcribed the spoken words. In two other cases there was only one researcher present who guided the participant with the session being recorded and transcribed afterwards by speech recognition. All sessions held as online video conferences lasted approximately one hour. The remaining session was held by the respective participant on their own, and the observations were submitted as written report. Two participants were involved in the design process as advisors, but had never actively used the system. The application was adapted to feedback from the first participant before the other four sessions were held.

The case study sessions started with a short introduction by the participant about themselves, including their research interests. A short demonstration of the main features by us followed. Then, the participants were asked to explore the application on their own while describing their thoughts ("thinking aloud"). We asked the participants to share their hypothesis about what response they expect to see from the network before trying out particular features of Perturber. Questions about the user interface were allowed and encouraged. During the last ten minutes of the session, the participants summarized their impressions of Perturber, told us what they liked and what they did not like, and gave us feedback about how the system could be improved for their needs. They were also

asked to share their insights gained from using Perturber, and if they found something out that they did not expect beforehand.

The participating researchers' fields of expertise were all related to model robustness and model interpretability. In particular, our participants self-described research topics were:

- P1: Understanding vision in humans and machines, with a special focus on Deep Learning interpretability and feature visualizations.
- P2: On the interface between psychophysics and deep learning, in particular understanding how object recognition differs between humans and machines.
- P3: Detection and interpretation of failure cases of computer vision models.
- P4: Learning more robust, safe, and verifiable machine learning models.
- P5: Designing interpretable deep learning models.

7.1.1 General Impressions and Feedback

The core concept of Perturber, interactively manipulating a scene while simultaneously observing neural network activations, was appreciated by all participants. They liked the immediate feedback, and some expressed surprise about the interactivity of the system. P1, P2 and P3 explicitly stated that the tool is beneficial for quickly generating new hypotheses. A particularly well appreciated feature was the morphing between cat and dog. P2 wondered where humans would make the transition to perceiving the object as "cat". P5 liked morphing texture and shape of the foreground object independently. P1 was generally impressed by the amount of features, but would have been overwhelmed by the interface without our guidance. After P1's session, we made GUI adaptions, separating the different visualization components more clearly. P2 liked that the classes are visualized with example images, because many classes, especially various dog races, might be unknown to the user. One participant suggested using Perturber as a teaching tool, and to make people aware of how far away deep learning models are from human-level intelligence. Apart from these general impressions, the participants had diverse suggestions to extend and adapt the system for their interests.

A recurring suggestion was to diversify or enable customization of the scene. P1 would have liked to use custom background images and foreground objects, while P3 would have found it helpful to experiment with more background options. he mentioned as an example an underwater scene, an unusual environment for the provided foreground models. P3 also suggested a functionality to rotate the background, and to experiment with more 3d models than the ones provided. Following these suggestions, we implemented functionality to select custom background images and 3d models from the user's local disk. Multiple participants expressed an interest for automatization features. P1 would have found it useful to extract meaningful circuits [CCG+20] automatically. She was aware that this was far beyond the scope of our tool. P2 stated that "at some point you'd want

to test it on hundreds or thousands of images", and P4 stated that a systematic evaluation of how various aspects of input perturbations affect different models' decisions would be beneficial, and that he would like to have a backend with the ability of performing a grid search.

Multiple participants commented on the activation / feature visualization area. While P2 stated that he was generally not knowledgeable about feature visualizations, P1 suggested dataset examples for visualizing intermediate neurons, and P3 would have liked to see multiple feature visualizations for each neuron to highlight their nature of responding strongly to diverse inputs (see "diversity" in Olah et al. 2017 [OMS17]). P1 liked the ability to scrub through the feature visualizations for various fine-tuning steps. She was surprised that feature visualizations generated on-the-fly for the "Edited Model" looked different compared to the pre-calculated feature visualizations, before we explained to them that there were slight implementation differences. P1 also stated that the comparatively slow generation of feature visualizations for the "Edited Model" was not distracting and gave them time to observe other visualizations in the mean time. P3 was surprised that feature visualizations for the adversarially trained model look more "cartoony" and found them easier to interpret because of their lack of texture. He was also surprised by his observation that some feature visualizations from the SIN trained model look more textured than the corresponding feature visualizations from the standard model.

7.1.2 Detailed Observations

During our sessions, the participants were asked to come up with hypotheses about model behaviour in response to input perturbations and to test them while simultaneously explaining their observations. In the following paragraphs, we will summarize these observations, grouped by the different types of input perturbations.

Camera transformations. P2 did not find it surprising that camera roll breaks the adversarially trained model, specifically a dog getting confused for a hen. He had recently done an experiment where he had observed that adversarially trained models were more sensitive to large scale transforms. Similarly, P4 observed that the adversarially trained model tends to misclassify the scene more often upon input viewpoint and structural variations than the standard trained model. P5 also observed that the adversarially trained model's classification output changes more than the standard trained model's classification output while rotating the camera. On the other hand, the adversarially trained model is less sensitive to the object's distance to the viewpoint, according to P4's observations.

Scene perturbations. P3 observed that a "Dog Relevant" feature map in Layer "mixed4d" did not peak at the full "Dog" position of the object morphing slider between "Dog" and "Cat", but at a value of approximately 5% "Cat". He also found that a blurred background generally makes the adversarially trained model behave less consistent, and therefore hypothesized that an adversarially trained model must rely more on

the background for making its prediction. We were able to confirm this hypothesis quantitatively, as shown in Section 7.3.2. A similar observation was made by P4, who found that perturbing the background image significantly alters the decisions made by the adversarially trained model, while the effect is less apparent in the standard trained model.

Post-processing perturbations. With post-processing perturbations, the expectations of participants who tried this feature were diverging. P2 was surprised that the SIN trained model held up well while blurring the image. He had hypothesized that it needs sharp lines because stylized images tend to contain relatively sharp, paint-stroke like structures. Similarly, P3 would have expected the SIN trained model to be more affected by blurring the image. On the other hand, P1 was surprised by the strong influence of frequency decomposition operations on the class prediction.

Adversarial attacks. Two participants made observations while using the adversarial attack module. P2 noticed that attacking the adversarially trained model produces an image that significantly differs from the original, while attacking the other models merely added high-frequency details. P5 focused on feature map activations during adversarial attacks. He noticed that activations in early layers changed very little compared to later layers by the attack. Also, he performed a targeted attack towards "badger" on the standard model with a car scene as original input. P5 then observed an increasing activation in previously non-activated fur-detecting neurons.

Combined perturbations. P2 hypothesized that there is an interaction between background and camera rotation. He tested the adversarially and standard trained models with a dog in front of the "street" background. The adversarially trained model was very brittle regarding small rotations with this unusual background, while the standard model held up better - an observation that supported their hypothesis. P3 found it interesting that the adversarially trained model confidently classified a close-up of the de-textured dog head as a hammerhead, and that a small rotation leads to very different classification scores. He did not further explore this behaviour. P3 also observed that with texture on the model, the SIN and adversarially trained models are more consistent under rotation when viewing a close-up of the dog's ears than without texture. He concluded that texture makes the classification more consistent.

Pruning and mixing. P5 was the only participant to use the pruning and mixing functionalities. When He pruned filter kernels by their magnitude in Layer "mixed5a", he was surprised about the strong effect that low-magnitude kernels had on the classification result, which was against their original intuition. While mixing weights from the standard model and the adversarially trained model layer by layer, he noticed that some layer combinations were more compatible (resulted in correct classfications) than others. Specifically, using early layers' weights up to "mixed3a" from one model and the other layers' weights from the second model worked much better than he had anticipated.

7.2 Our Findings

During the development of Perturber, we often made interesting discoveries while testing the application and its latest features ourselves. One of these findings stands out among the others, and to the best of our knowledge has not been published before. We made the following observation while mixing layers from the standard and from the adversarially trained model with the weight editing module as described in Section 5.4. We noticed that an adversarial attack on a model using early layer weights from the adversarially trained model, for instance up to including "mixed3a", in combination with weights from the standard trained model for the rest of the layers, leads to an image that shows remarkable low frequency structure. The appearance of the attack more closely resembled an attack generated for an adversarially trained model than for a standard trained model, as shown in Figure 7.1.

As Tsipras et al. have shown [TSE+19], attacks on adversarially trained models generally exhibit low frequency structure, which led us to suspect that our mixed model might have improved robustness just through the use of early layer weights from an adversarially trained model. This could imply a cheaper method to strengthen a model's adversarial robustness than full adversarial training, by re-using adversarially pre-trained layers.

We subsequently conducted an experiment where we fixed either just the first or both, the first and the second group (out of four) of residual blocks [HZRS16] of an adversarially trained ResNet 18 model, and then trained the remaining weights on ImageNet classification in a standard, non-adversarial way. We compared both obtained models to a standard-trained ResNet 18 under varying-strength adversarial attacks on the ImageNet validation set. We found that the models with the fixed early layer weights from the adversarially trained model consistently outperformed the standard model under all attack magnitudes greater than zero. The model with just the first group's weights fixed even outperformed the standard trained model. The results of our measurements are shown in Figure 7.2.

7.3 Quantitative Measurements for Case Studies

To find out if Perturber can ideed be useful for generating hypotheses that can withstand quantitative evaluation, we performed quantitative measurements to verify some of the discoveries by users described in Section 7.1. To test the generalizability of the users' observations, we performed these measurements using different models than the ones used in the online tool. We used the pre-trained ResNet 50 from the *torchvision* library of *PyTorch* [PGM+19] as standard model. To verify hypotheses that involved different behaviour between standard and adversarially trained models, we used weights of an adversarially trained version of the same ResNet 50 from the *robustness* library [EIS+19] (ResNet 50 ImageNet ϵ 3/255 under L_2-norm).

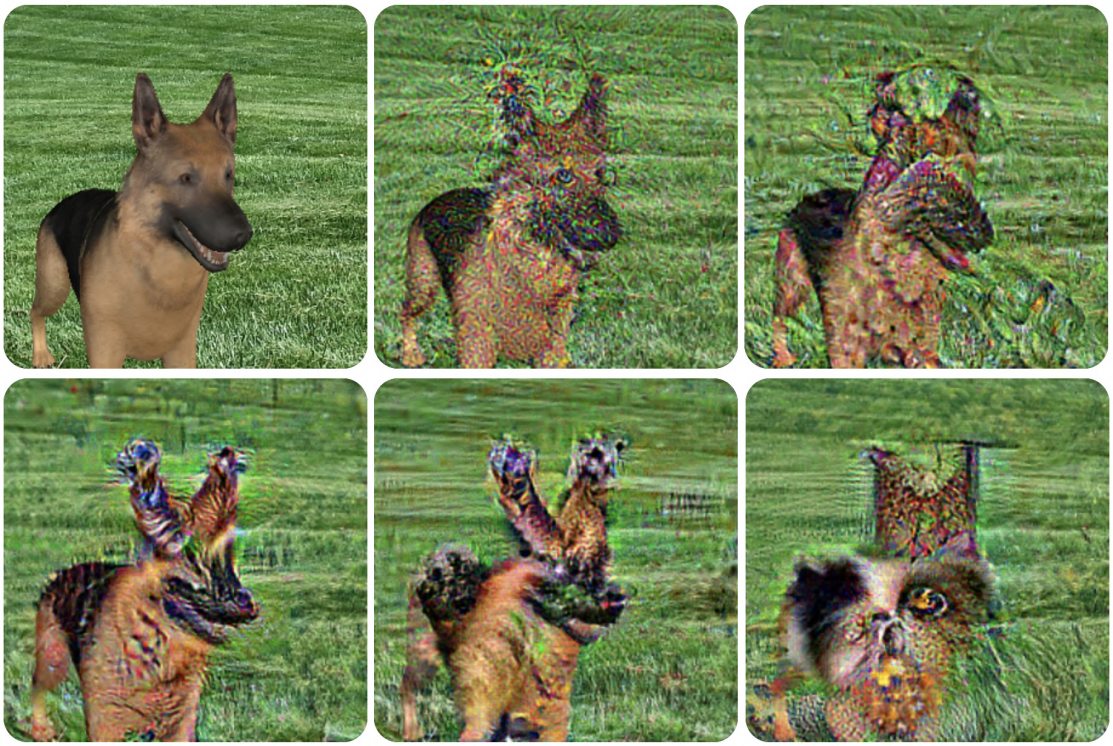


Figure 7.1: Original image (top left); Standard model adversarial attack (top center); Mixed model adversarial attacks with successively more layers with weights from the adversarially trained model: Up to "conv2d1" (top right); Up to "mixed3a" (bottom left); Up to "mixed4c" (bottom center); All layers (bottom right). All attacks are untargeted.

7.3.1 Are Adversarially Trained Models More Sensitive to Viewpoint Changes?

First, we verified if the adversarially trained model is more affected by camera transformations than the standard model, as reported by P2, P4, and P5. Specifically, we tested behaviour under camera rotation. We created a synthetic dataset by rendering the four 3d models available in Perturber from seven angles around the yaw axis ({-70°, -46.7°, -23.3°, 0°, 23.3°, 46.7°, 70°}, as shown in Figure 7.3 (b)), two pitch angles ({0°, -16.7°}), and two distances of the camera to the object, as shown in Figure 7.3 (a), resulting in 28 views for each of the four 3d models. The generated data set is publicly available at https://github.com/stefsietz/perturber.

To measure how much the predictions vary between camera angles, we defined a prototype view with a yaw angle of -23.3°, serving as a reference which is neither overly frontal or overly from the side. We rendered each of the four models from this yaw angle and with each of the four pitch / distance combinations, as shown for two models in Figure 7.3 (a). For each of the 16 resulting prototypes, the logits of the top-10 classes served as "ground truth". We then compared the logits of the other views to the top-10 prototype logits. For

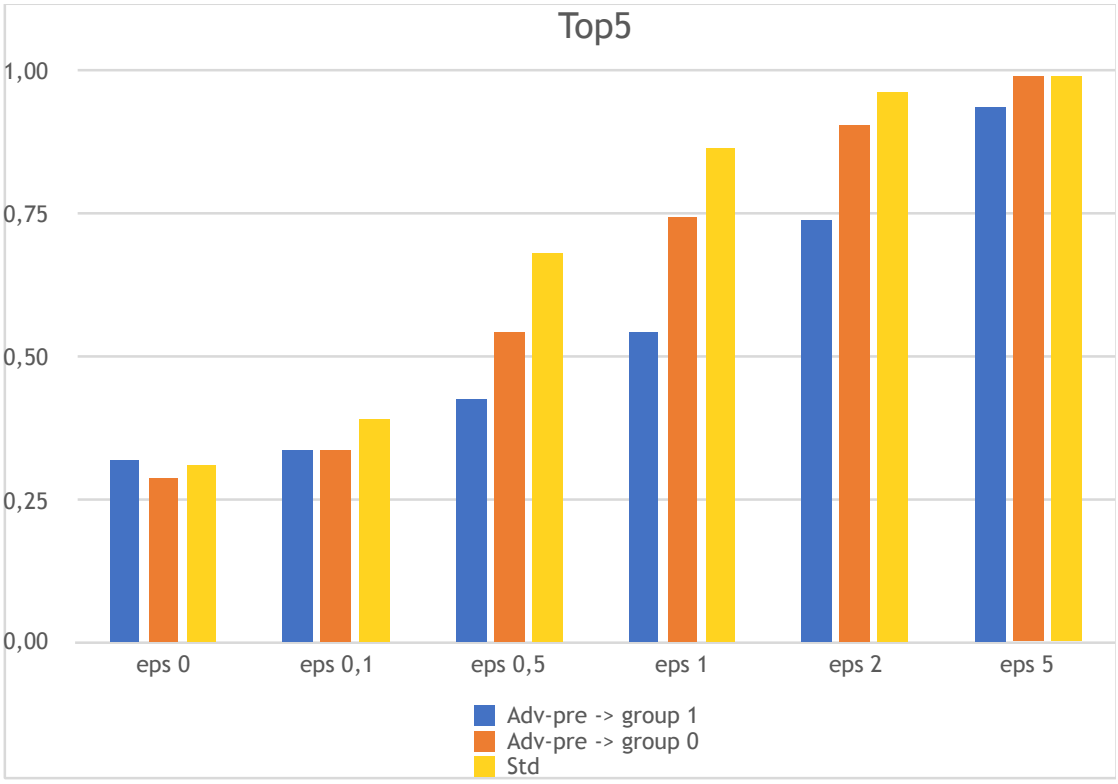


Figure 7.2: Top-5 error rates on the ImageNet validation set for a standard ResNet 18 model (yellow) and two ResNet 18 models with adversarially pre-trained weights up to including group 0 (orange) and group 1 (blue).

every yaw variation, we computed the Euclidean distance of the logit values of the top-10 ground truth classes vector between each yaw-angle varied image and the prototype view. We normalized the Euclidean distance of the top 10-logits by the standard deviation of the total logit vector and averaged the score per object.

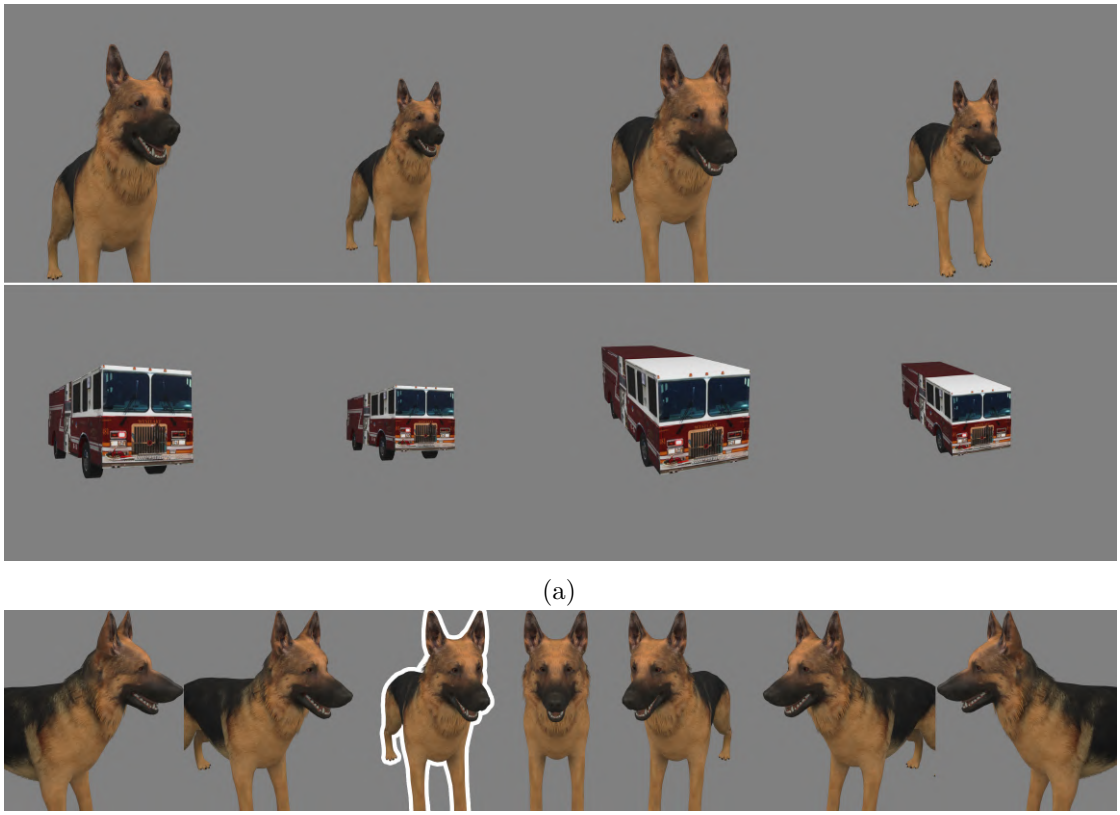
Formally, this "fluctuation score" f_p can be expressed in the following way:

$$\mathbf{d}_y = \mathbf{l}^*_{c^*_{10}} - \mathbf{l}^y_{c^*_{10}}, \tag{7.1}$$

$$f_p = \sum_y \frac{\sqrt{\mathbf{d}_y \cdot \mathbf{d}_y}}{std(\mathbf{l}^*_{1000})}, \tag{7.2}$$

where $\mathbf{c}^*_{10}$ are the top 10 predictions of the prototype view, $\mathbf{l}^y_{c^*_{10}}$ are their logits from yaw variation y, $\mathbf{l}^*_{c^*_{10}}$ are their logits from the prototype view itself and $\mathbf{l}^*_{1000}$ is the total logit vector of the prototype view.

The resulting fluctuation scores are considerably higher for the adversarially trained model except for the race car object (Figure 7.4). This is strong evidence that the



(a)

(b)

Figure 7.3: Four prototype views for combinations of slightly varied camera pitch angles and camera distances, shown for the dog and fire truck models (a). For each of those prototype views, we generate six additional yaw angle variations. The bottom image (b) shows those variations for the dog view outlined in white, which corresponds to the second view from right in (a).

adversarially trained model is indeed more vulnerable to yaw rotations of the main scene object.

7.3.2 Are Adversarially Trained Models More Sensitive to Background Changes?

Our second quantitative measurement investigated the adversarially trained model's sensitivity to background changes, which was observed by P2 and P5. We performed the *Background Challenge* by Xiao et al. [XEIM20], where images with replaced backgrounds are used to test the background's role as a feature for the model's classification result. In this testing scenario, random guessing would result in 11.1% accuracy [XEIM20]. The adversarially trained model only achieved an accuracy of 12.3%, barely exceeding random guessing, while the standard model achieved 22.3% accuracy. This result verifies that the

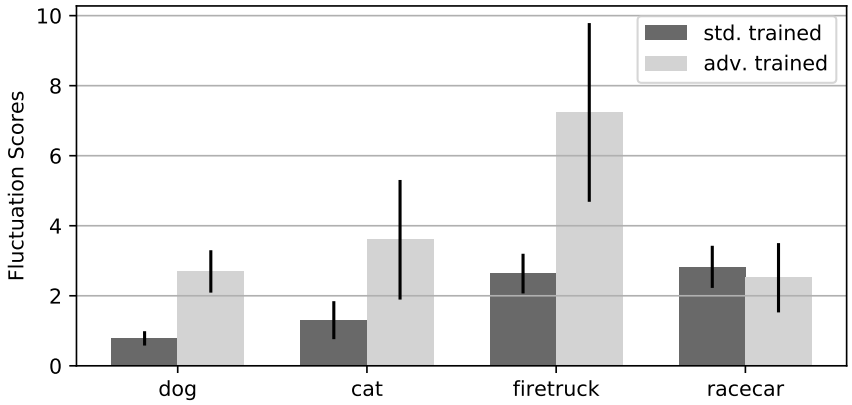


Figure 7.4: Mean yaw fluctuation scores for the standard model and the adversarially trained model for the four 3d models. Error bars represent standard deviation.

adversarially trained model is more dependant on backgrounds than the standard model.

7.4 Performance Measurements

While designing Perturber, instantaneous visual feedback was always a top priority. We continuously assessed integrated visualizations regarding interactivity and removed several of those visualizations after they failed to meet this requirement. Examples of visualizations that we had implemented but found to be too slow for our instantaneous feedback requirement are Grad-CAM [SCD+17] and a visualization of the class probability gradient with respect to the input image.

We measured Perturbers performance by recording framerates on two representative notebook types and with various visualization configurations. The notebooks used for our performance benchmarks were a MacBook Pro 13" 2018 with Intel Iris Plus Graphics 655 (MBP) and a Gigabyte AORUS 15G Gaming Notebook with an NVIDIA GeForce GTX 2080 Super GPU (AORUS). Figure 7.5 shows the recorded framerates. Clearly, the GPU has a strong influence on the frame rate, but also the enabled visualizations have a significant impact. The prediction view (Section 5.5) requires a forward pass through the whole CNN, while the neuron activation view (Section 5.3) only requires inference up to the selected layer. Notably, the AORUS notebook, while having a significantly faster GPU than the MBP, only has a slightly higher framerate than the MBP when the former shows the prediction view and the later does not.

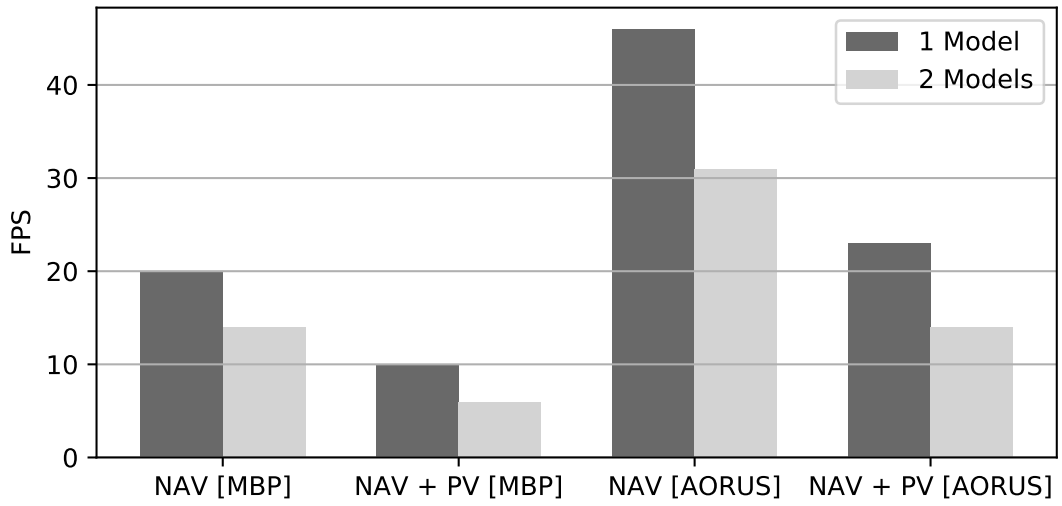


Figure 7.5: Performance benchmarks for four different output configurations, measured on two machines (MBP or AORUS): neuron activation view (NAV) only, or neuron activation view in combination with prediction view (NAV+PV) visualized for one or two visualized models concurrently.

CHAPTER 8

Conclusion and Future Work

In this final chapter, we will briefly summarize and recapitulate the presented work. Then we will look at shortcomings of the current version of our application, and at potential future work that might address these shortcomings.

8.1 Summary

In our introduction, we presented the issues of interpretability and robustness of neural networks. What features do neurons respond to? How are features different between standard- and robustly trained models? How can we visualize them? We explained state of the art feature visualization methods, the concept of adversarial examples and their relationship. We highlighted the connection between adversarial examples and the strong reliance of standard CNNs on high frequency features. While adversarial attacks on standard trained models are mostly high-frequency perturbations, attacking an adversarially trained model leads to globally structural changes, as Madry et al. showed [MMS+18]. This highly interesting phenomenon motivated our first hypothesis:

H1: Feature visualization can help to understand the difference of learned features between standard and robust training.

In Chapter 4, we then presented an extensive collection of experiments designed to verify H1. We proposed three initial questions in Section 4.1 and we subsequently answered them with feature visualization techniques.

- *When during training do feature detectors diverge in standard versus adversarial training?* In Section 4.1.1, we showed that feature detectors diverge right from the beginning, by comparing various adversarial training runs with a standard training run, all other hyper-parameters being fixed.

- *How do feature detectors develop during training in general, when viewed through the lens of feature visualization?* We confirmed visually that early feature detectors converge earlier than later feature detectors. This behaviour has previously been shown quantitatively by Raghu et al. [RGYSD17]. By visualizing this phenomenon, we give researchers the ability to build intuition about feature convergence during the CNN training process. Additionally, the fact that feature convergence during training is observable through feature visualizations, strengthens their validity in revealing the details of neuron-activating patterns.
- *What is the influence of the perturbation epsilon? Are there any feature visualization characteristics that correlate with perturbation epsilon?* To answer this question, we trained eight models adversarially, varying the epsilon value between zero (standard training) and 1.0. We then generated feature visualizations for each model at various layers and visually compared them against each other. We found a clear correlation between low image frequencies and epsilon value, although with a visible saturation of low frequencies at an epsilon value of 0.33.

After these experiments, in which we investigated the difference between standard and adversarial training from scratch, we looked at adversarial fine-tuning. By generating feature visualizations for checkpoints during an adversarial fine-tuning process, we were able to observe the features changing towards a representation with a stronger low-frequency focus. We compared these feature transitions to their counterparts derived from fine-tuning on Stylized ImageNet, which didn't exhibit such a striking transformation. This comparison highlighted the unique characteristics of adversarial training.

The formulation of H1 makes a clear confirmation or rejection of the hypothesis hard. We certainly have not fully understood the difference between standard and adversarial training, therefore claiming that feature visualization can help in doing so is not possible. Still, we have shed light onto some important aspects of their difference. Particularly the comparison of feature visualizations between adversarial fine-tuning and Stylized ImageNet fine-tuning highlights the distinctive role of adversarial training: Even though training on Stylized ImageNet alleviates the bias towards texture features compared to training on standard ImageNet, our results indicate that the obtained feature representation still exhibits similar high-frequency dependence.

Our second hypothesis H2 was targeted at a more general notion of robustness.

H2: An application, that allows interactively applying and combining a large palette of image perturbations while inspecting intermediate- and output activations of a single or multiple CNNs, can help to investigate the CNNs robustness.

We designed and implemented such an application, incorporating feedback from deep learning and visualization experts. We integrated models and feature visualizations which had been generated during our preliminary experiments, allowing the meticulous inspection of multiple checkpoints from adversarial- and Stylized ImageNet fine-tuning.

The visual analytics application enables the user to interactively perturb the input of a CNN by manipulating a 3d scene. Apart from 3d scene parameters such as camera view, object morphing, texture- and lighting influence, our application provides numerous post processing effects inspired by recent work on model robustness. Additionally, we enable users to perturb the model itself, by mixing weights from two models on a layer-by-layer basis, and by pruning weight kernels by channel magnitude. Notably, the visual analytics application does all heavy computation on the client side and therefore can be served as a web application to millions of users without costly server infrastructure.

To verify H2, we conducted five expert case studies which we presented in Section 7.1. The results from these case studies consist of a large variety of anecdotal observations which we believe can be beneficial for intuition-building. More importantly, they contain two observations which we were able to verify quantitatively. In combination with our own observations made during interactive weight mixing experiments, they provide strong evidence in favor of H2.

8.2 Limitations and Future Work

In the following paragraphs we will discuss various limitations of our work. Some of them have been observed and described by our case study participants, some of them are our own observations. As our visual analytics application has been optimized for usability and minimum complexity, we even describe some potential features here that were previously implemented but then omitted for reducing complexity. After looking at limitations and potential improvements of our visual analytics application Perturber, we will briefly discuss future work that can extend experiments and further solidify results that have been presented in this thesis.

A major limitation of our visual analytics application is the tight integration of the Inception V1 architecture. Extending the application to other models is possible but requires several hours of manual work. We had integrated ResNet 18 during the development but switched to Inception V1 because of the large amount of feature categorization already done by Olah et al. [OCS+20], as part of the Circuits project. Omitting the activation / feature visualization module would greatly reduce the effort necessary to adapt the application to a new model. The feature visualizations also require the majority of the computational pre-processing budget, as they are generated for all neurons at multiple training checkpoints. Another possibility would therefore be to restrict the model comparison to fully trained models instead of providing multiple training checkpoints. In fact, this would be the only way to enable the proper integration of pre-trained models, where the user does not have the opportunity to save arbitrary checkpoints during training. Even though the case study participants generally liked the activation / feature visualization model, we believe that it is of less importance than the output module, and that the benefits of an interface without it could potentially outweigh its drawbacks.

In summary, the main benefits of a version of Perturber without multiple checkpoints, and without the activation / feature visualization module would include:

- Relatively easy integration of arbitrary classifier models - even non CNN architectures.
- Direct comparison of models with different architecture.

The main drawbacks would be:

- No direct juxtaposition of hidden-layer activations from multiple models.
- No feature visualizations.
- No weight mixing capabilities.

Such an application might lack some of the functionality present in our work, but it could enable a more diverse, more general investigation and comparison of model robustness.

Several of our case study participants requested a form of systematic search over the perturbation parameters. This would certainly be possible to integrate in some form, although searching over more than two parameters would be prohibitively slow. During Perturber's development, we experimented with a two dimensional grid search with selectable resolution and perturbation parameters. This module sequentially generated images for the respective parameter combinations and recorded the activations for all units of a layer. It then was able to display them in a similar way to how OpenAI Micropscope's "synthetic tuning curves" display unit responses - the input images were arranged in a grid for each unit, and each input image was saturated according to the response it caused in the respective unit. We decided to omit this feature in the final interface design for visual clarity and improved user experience. Integrating it would require an additional row of square tuning curve images beneath the feature visualizations for each neuron, as well as a menu to configure and initiate the grid search. As mentioned in Section 3.2, 3DB [LSI+21] is a framework that facilitates such a systematic search.

Another form of automatic search requested by participants was to compute numerical partial derivatives of the current output classification score with respect to the perturbation parameters. This would require two additional image generations and model evaluations per perturbation parameter for which the partial derivative has to be computed. The numerical differentiation step size would be a hyperparameter to be determined for each parameter. While we did not explore the feasibility of implementing this feature in our application, we can certainly see a lot of value for the user to have this information displayed at each parameter control, to help choose which parameters to change.

One of our results presented in Section 7.2 indicates that fixed, pre-trained early layers from an adversarially trained model provide a non-trivial improvement of adversarial robustness when training a standard model. This finding might help in the development of training optimizations for adversarial training, leading to some degree of adversarial robustness at a massively reduced training cost. Our experiments were restricted to the ResNet 18 architecture and two different transfer-learning configurations. Such a small number of experiments can only serve as anecdotal evidence and therefore further investigation of this phenomenon would be desirable.

To conclude, let us very briefly reiterate the main contributions of the presented work:

- We collected evidence for a causal connection between a model's behaviour and the visual appearance of their feature visualizations.
- We developed a visual analytics application allowing for interactive model probing under various scene and image perturbations in coordination with domain experts.
- We then conducted case studies with five experts, showing that novel observations can be made by using our application.
- We tested and confirmed two of the experts' observations, as well as one of our own observations, quantitatively.

We firmly believe that our visual analytics application serves as a valuable proof of concept on the intersection between visual analytics and neural network interpretability with respect to robustness. The described limitations suggest potential for further improvement with respect to automatization, adaptability and user experience. The core concept of interactively manipulating and perturbing a scene, serving as an input to a model under inspection, has been shown to be useful.

We further believe that Perturber, once published on the web, will serve many researchers and students as a useful playground for experimentation and intuition building. Perturber's unique and unusual ability among visual analytics tools is to perform expensive computations on the client's GPU. Therefore it is inexpensive to deploy on the web for wide access. We hope that this property will motivate open source contributors to improve the input scene of Perturber, so that it can benefit from ongoing and future developments in photorealistic rendering.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ACW18] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, July 2018.

[AZL20] Zeyuan Allen-Zhu and Yuanzhi Li. Feature purification: How adversarial training performs robust deep learning. *arXiv preprint arXiv:2005.10190*, 2020.

[BA87] Peter J. Burt and Edward H. Adelson. The Laplacian Pyramid as a Compact Image Code. In Martin A. Fischler and Oscar Firschein, editors, *Readings in Computer Vision*, pages 671–679. Morgan Kaufmann, San Francisco (CA), 1987.

[BG06] Daniel A Butts and Mark S Goldman. Tuning curves, neuronal variability, and sensory coding. *PLoS biology*, 4(4):e92, 2006.

[BZS+20] Judy Borowski, Roland Simon Zimmermann, Judith Schepers, Robert Geirhos, Thomas SA Wallis, Matthias Bethge, and Wieland Brendel. Exemplary natural images explain cnn activations better than state-of-the-art feature visualization. In *International Conference on Learning Representations*, 2020.

[CAS+19] Shan Carter, Zan Armstrong, Ludwig Schubert, Ian Johnson, and Chris Olah. Activation Atlas. *Distill*, 2019.

[CCG+20] Nick Cammarata, Shan Carter, Gabriel Goh, Chris Olah, Michael Petrov, and Ludwig Schubert. Thread: Circuits. *Distill*, 5(3):e24, March 2020.

[CGC+20] Nick Cammarata, Gabriel Goh, Shan Carter, Ludwig Schubert, Michael Petrov, and Chris Olah. Curve detectors. *Distill*, 2020.

[CPCS20] D. Cashman, A. Perer, R. Chang, and H. Strobelt. Ablate, Variate, and Contemplate: Visual Analytics for Discovering Neural Architectures. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):863–873, January 2020.

[CW17] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, Los Alamitos, CA, USA, may 2017.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[CZM+19] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019.

[DDS+09] 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.

[DT17] Terrance DeVries and Graham W Taylor. Improved regularization of convolutional neural networks with cutout. *arXiv preprint arXiv:1708.04552*, 2017.

[EBCV09] Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. Visualizing Higher-Layer Features of a Deep Network. *Technical Report, Univeristé de Montréal*, January 2009.

[EIMX20] Logan Engstrom, Andrew Ilyas, Aleksander Madry, and Kai Xiao. Noise or signal: The role of backgrounds in image classification. `https://gradientscience.org/background/`, 2020. (Accessed on 12/02/2021).

[EIS+19] Logan Engstrom, Andrew Ilyas, Hadi Salman, Shibani Santurkar, and Dimitris Tsipras. Robustness (Python Library). `https://github.com/MadryLab/robustness`, 2019. (Accessed on 10/12/2020).

[GEB16] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[GH19] Justin Gilmer and Dan Hendrycks. A discussion of 'adversarial examples are not bugs, they are features': Adversarial example researchers need to expand what is meant by 'robustness'. *Distill*, 2019.

[GMP+17] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, volume 2, pages 2672–2680, 2014.

[GRM+18] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2018.

[GSS15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, Conference Track Proceedings*, 2015.

[Har15] Adam W. Harley. An Interactive Node-Link Visualization of Convolutional Neural Networks. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Ioannis Pavlidis, Rogerio Feris, Tim McGraw, Mark Elendt, Regis Kopper, Eric Ragan, Zhao Ye, and Gunther Weber, editors, *Advances in Visual Computing*, pages 867–877. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.

[HBM+21] Dan Hendrycks, Steven Basart, Norman Mu, Saurav Kadavath, Frank Wang, Evan Dorundo, Rahul Desai, Tyler Zhu, Samyak Parajuli, Mike Guo, et al. The many faces of robustness: A critical analysis of out-of-distribution generalization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8340–8349, 2021.

[HD18] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *International Conference on Learning Representations*, 2018.

[HKPC19] F. Hohman, M. Kahng, R. Pienta, and D. H. Chau. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE*

Transactions on Visualization and Computer Graphics, 25(8):2674–2693, August 2019.

[HMC+20] Dan Hendrycks, Norman Mu, Ekin D. Cubuk, Barret Zoph, Justin Gilmer, and Balaji Lakshminarayanan. AugMix: A simple data processing method to improve robustness and uncertainty. *Proceedings of the International Conference on Learning Representations*, 2020.

[HPRPC20] Fred Hohman, Haekyu Park, Caleb Robinson, and Duen Horng Polo Chau. Summit: Scaling Deep Learning Interpretability by Visualizing Activation and Attribution Summarizations. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1096–1106, January 2020.

[HZB+21] Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. Natural adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15262–15271, 2021.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[JB17] Jason Jo and Yoshua Bengio. Measuring the tendency of CNNs to Learn Surface Statistical Regularities. *arXiv:1711.11561*, November 2017. arXiv: 1711.11561.

[Kha20] Vaibhav Khandelwal. The architecture and implementation of vgg-16. `https://pub.towardsai.net/the-architecture-and-implementation-of-vgg-16-b050e5a5920b`, 2020.

[KPN16] Josua Krause, Adam Perer, and Kenney Ng. Interacting with predictions: Visual inspection of black-box machine learning models. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 5686–5697, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges,

L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[KTC+19] M. Kahng, N. Thorat, D. H. Chau, F. B. Viégas, and M. Wattenberg. GAN Lab: Understanding Complex Deep Generative Models using Interactive Visual Experimentation. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):310–320, January 2019.

[LBM+16] Sebastian Lapuschkin, Alexander Binder, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. The LRP Toolbox for Artificial Neural Networks. *Journal of Machine Learning Research*, 17(114):1–5, 2016.

[LCJ+19] Dongyu Liu, Weiwei Cui, Kai Jin, Yuxiao Guo, and Huamin Qu. DeepTracker: Visualizing the Training Process of Convolutional Neural Networks. *ACM Transactions on Intelligent Systems and Technology*, 10(1):1–25, January 2019.

[LHA+20] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas A Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. *Nature Machine Intelligence*, 2(10):642–652, 2020.

[LLL+18] Shusen Liu, Zhimin Li, Tao Li, Vivek Srikumar, Valerio Pascucci, and Peer-Timo Bremer. Nlize: A perturbation-driven visual interrogation tool for analyzing and interpreting natural language inference models. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):651–660, 2018.

[LLS+18] M. Liu, S. Liu, H. Su, K. Cao, and J. Zhu. Analyzing the Noise Robustness of Deep Neural Networks. In *2018 IEEE Conference on Visual Analytics Science and Technology*, pages 60–71, October 2018.

[LLUZ16] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 4905–4913, 2016.

[LSI+21] Guillaume Leclerc, Hadi Salman, Andrew Ilyas, Sai Vemprala, Logan Engstrom, Vibhav Vineet, Kai Xiao, Pengchuan Zhang, Shibani Santurkar, Greg Yang, Ashish Kapoor, and Aleksander Madry. 3db: A framework for debugging computer vision models. In *Arxiv preprint arXiv:2106.03805*, 2021.

[MFH+20] Y. Ma, A. Fan, J. He, A. R. Nelakurthi, and R. Maciejewski. A Visual Analytics Framework for Explaining and Diagnosing Transfer Learning Processes. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1385–1395, 2020.

[Mil95] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[MMS+18] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[MS18] Alexander Madry and Ludwig Schmidt. A brief introduction to adversarial examples. http://gradientscience.org/intro_adversarial/, 2018. (Accessed on 02/12/2020).

[MV15] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5188–5196, 2015.

[NCB+17] Anh Nguyen, Jeff Clune, Yoshua Bengio, Alexey Dosovitskiy, and Jason Yosinski. Plug & Play Generative Networks: Conditional Iterative Generation of Images in Latent Space. In *2017 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3510–3520, July 2017.

[NDY+16] Anh Nguyen, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. *Advances in Neural Information Processing Systems*, 29:3387–3395, 2016.

[NQ17] Andrew P Norton and Yanjun Qi. Adversarial-Playground: A visualization suite showing how adversarial examples fool deep learning. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–4, 2017.

[NYC15] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *2015 IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[OBLS14] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1717–1724, June 2014.

[OCS+20] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 2020.

[OMS17] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature Visualization. *Distill*, 2(11):e7, November 2017.

[Ope] OpenAI Microscope. `https://microscope.openai.com/models`. (Accessed on 10/12/2020).

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PHVG+18] Nicola Pezzotti, Thomas Höllt, Jan Van Gemert, Boudewijn P.F. Lelieveldt, Elmar Eisemann, and Anna Vilanova. DeepEyes: Progressive Visual Analytics for Designing Deep Neural Networks. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):98–108, January 2018.

[PMJ+16] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy*, pages 372–387, 2016.

[PMW+16] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy*, pages 582–597, 2016.

[RGYSD17] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: singular vector canonical correlation analysis for deep learning dynamics and interpretability. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6078–6087, 2017.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.

[RSG16] Marco Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 97–101, June 2016.

[RZT18] Amir Rosenfeld, Richard Zemel, and John K. Tsotsos. The Elephant in the Room. *arXiv:1808.03305 [cs]*, August 2018. arXiv: 1808.03305.

[SCD+17] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In *2017 IEEE International Conference on Computer Vision*, pages 618–626, October 2017.

[SCS+17] Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viégas, and Martin Wattenberg. Direct-Manipulation Visualization of Deep Networks. *arXiv:1708.03788*, August 2017.

[SKC+20] Róbert Szabó, Dániel Katona, Márton Csillag, Adrián Csiszárik, and Dániel Varga. Visualizing Transfer Learning. *arXiv:2007.07628 [cs]*, July 2020. arXiv: 2007.07628.

[SLB+21] Stefan Sietzen, Mathias Lechner, Judy Borowski, Ramin Hasani, and Manuela Waldner. Interactive analysis of cnn robustness. *Computer Graphics Forum (Proceedings of Pacific Graphics 2021)*, 40(7):253–264, 2021.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[SSSEA20] T. Spinner, U. Schlegel, H. Schäfer, and M. El-Assady. explAIner: A Visual Analytics Framework for Interactive and Explainable Machine Learning. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1064–1074, January 2020.

[STA+19] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, and Martin Wattenberg. TensorFlow.js: Machine Learning for the Web and Beyond. *arXiv:1901.05350 [cs]*, February 2019. arXiv: 1901.05350.

[STK+17] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. SmoothGrad: removing noise by adding noise. *arXiv:1706.03825 [cs, stat]*, June 2017. arXiv: 1706.03825.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328, 2017.

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Workshop at International Conference on Learning Representations*, 2014.

[SW19] Stefan Sietzen and Manuela Waldner. Interactive feature visualization in the browser. *Proceedings of the Workshop on Visualization for AI explainability*, 2019.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[SZS+14] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations*, 2014.

[TSE+19] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. In *International Conference on Learning Representations*, 2019.

[W+16] Yuxin Wu et al. Tensorpack. `https://github.com/tensorpack/`, 2016. (Accessed on 01/06/2020).

[WPB+20] J. Wexler, M. Pushkarna, T. Bolukbasi, M. Wattenberg, F. Viégas, and J. Wilson. The What-If Tool: Interactive Probing of Machine Learning Models. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):56–65, January 2020.

[WSW+18] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg. Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12, January 2018.

[WTS+20a] Zijie J. Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng (Polo) Chau. Bluff: Interactively deciphering adversarial attacks on deep neural networks. In *IEEE Visualization Conference*, 2020.

[WTS+20b] Zijie J Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Polo Chau. Cnn explainer: Learning convolutional neural networks with interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1396–1406, 2020.

[XEIM20] Kai Yuanqing Xiao, Logan Engstrom, Andrew Ilyas, and Aleksander Madry. Noise or signal: The role of image backgrounds in object recognition. In *International Conference on Learning Representations*, 2020.

[XEQ18] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[YCFL15] Jason Yosinski, Jeff Clune, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. In *Workshop on Deep Learnin, International Conference on Machine Learningg*, 2015.

[YHO+19] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6023–6032, 2019.

[YLS+19] Dong Yin, Raphael Gontijo Lopes, Jonathon Shlens, Ekin D Cubuk, and Justin Gilmer. A fourier perspective on model robustness in computer vision. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 13276–13286, 2019.

[ZCDLP18] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations*, 2018.

[ZHP+17] Haipeng Zeng, Hammad Haleem, Xavier Plantaz, Nan Cao, and Huamin Qu. Cnncomparator: Comparative analytics of convolutional neural networks. *arXiv preprint arXiv:1710.05285*, 2017.

[ZLK+17] 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*, 40(6):1452–1464, 2017.

[ZZ19] Tianyuan Zhang and Zhanxing Zhu. Interpreting adversarially trained convolutional neural networks. In *International Conference on Machine Learning*, pages 7502–7511, 2019.