Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Deep Introspection: Explaining Neural Networks Through Feature Occlusion and Synthesis

Author:
Oliver Norton

Supervisor:
Dr. Bernhard Kainz

Second Marker:
Dr. Ben Glocker

Abstract

Neural networks, and in particular, convolutional neural networks have been successful in a wide variety of visual tasks such as image classification, object detection and image captioning. However, the main problem with these models as that they act as a black-box. We may be able to get accurate predictions from them, but we largely have no idea how they arrived at those predictions. This can be a problem in situations which require a large amount of trust in the results e.g. medical diagnosis. With the new GDPR regulations that came into effect in May 2018, consumers now have a "right to explanation". Since many businesses are starting to adopt models such as neural networks into their business processes, it is essential for them to have some way of explaining their models' decisions.

With these problems in mind, we propose two separate tools that help to explain convolutional neural network classifications. The first one extracts regions of the image that it considers important for the classification and allows the user to occlude them, so they can analyse how the network adapts to missing information. The second tool attempts to synthesise these features so that we can understand what the network "sees" as that feature. We have also developed a web application that allows users to upload their own models and images to use these tools.

Acknowledgements

I would first like to thank my supervisor Dr. Bernhard Kainz for his valuable advice and support throughout this project. Our initial discussions were very valuable to definite the initial direction and objectives of the project.

I would also like to thank my friends Alan for giving me advice on my application UI and Mickey for our discussions of ideas for our projects.

Finally, I would like to thank my parents and the rest of my family for their love and support throughout my four years at Imperial.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial neural networks have recently revolutionised various applications such as image recognition, speech transcription and natural language processing. However, researchers who develop these models and users alike do not understand exactly how these models come to their predictions. This means that the user cannot build trust with a system despite their high accuracy and researchers cannot fully analyse their models in order to make improvements.

Though there have been various techniques developed to explain neural network decisions and visualising what neural networks learn, there is not currently a model-agnostic tool developed in order to incorporate these explanation techniques. In this project, we present a feature exploration tool for deep neural network models. This tool implements two different feature exploration techniques: salient regions and region synthesis and is also allows various neural network models to be analysed and provide visual explanations for decisions.

1.1 Motivation

In recent years, deep neural networks and in particular convolutional neural networks have made a huge impact in various applications such as image recognition [KSH12, SLJ+15, HZRS16], object segmentation [BHC15, LSD15] and natural language processing. These models, originally modelled after biological neural networks in the human brain, link layers of computing units called neurons together. Though each neuron itself performs fairly simple operations on inputs, the way these neurons are linked together with outputs of one neuron serving as the inputs of others allows very complex functions to be represented. With the appropriate architecture, a large enough dataset and long enough time for training it appears that nearly any non-dynamic domain can be tackled using neural networks.

However, due to their multilayer and nonlinear structure, these models are considered "black boxes" i.e. difficult for a human to understand *how* and *why* a particular model came to its decision from a particular input. This can mean that despite having a high performance in a particular domain, their prediction may not be trusted by the user. Though this can just be inconvenient in some problem domains (e.g. automatic face tagging in Facebook [Vin17]), there are some domains where the correct decision is particularly important ch as a medical diagnosis where this is unacceptable. For example, [CLG+15] presents a system that predicts pneumonia risk but arrives at totally wrong conclusions. Due to statistical bias, it concluded that patients with asthma or heart problems were less likely to die from pneumonia. Since it is difficult for even experienced researchers to understand how exactly deep neural networks reach their decisions, weakness analysis for example is not possible. Therefore, most improvements in both training neural networks and finding new novel architectures is through trial-and-error [Wel17].

As well as explaining how a system reached a corrected prediction it is just as, if not more so, for a system to explain its reasoning. An interesting example for image recognition tasks is the case of the leopard print sofa [Khu15]. To a human, the object in Figure 1 is obviously some kind of sofa or chair and not a leopard. However, to a lot of neural network architectures, this would be classified instead as a leopard or jaguar. This, again, is due to a statistical bias in

Figure 1.1: A sofa instantly recognised by humans but fools many CNN architectures [Khu15]

the ImageNet[DDS+09] dataset where there is a much larger proportion of leopards and jaguars compared to leopard print sofas. Therefore, the network reasons that if it recognises a spotty pattern, it's better off guessing a leopard or jaguar than a sofa. There are also generated adversarial examples that will look identical to humans but fool a neural network with high probability [MDFF16, DSZB17] and others that look unnatural but identified by neural networks with high probability [NYC15] which will be discussed further in the background section.

Artificial neural networks are being used in more and more domains and architectures are also becoming more complex. However, there are still significant advancements that need to be made for getting neural networks to explain their reasoning. With that and EU regulations on the "right to explain" being made into the law in 2018 [GF16], this is an important field to look into. On top of the legal issue, getting neural networks to explain their decisions also helps build trust in them and helps researchers improve existing models [Wel17].

1.2 Objectives

Having neural networks be able to explain the rationality behind their decisions benefits both developers and researchers alike. For developers who use neural networks in their applications, it is beneficial for finding possible biases and other problems in their datasets as well as edge cases for their particular problem. For researchers in neural networks, explainability allows them to analyse the weaknesses of architectures and potentially find problems with their dataset. The overall goal is to produce an application that helps these users by explaining how and why their neural network models made a particular prediction for a particular image. Therefore there are several main objectives for this project:

1. **Easy to use application** As this will be used by a variety of users, some who will not be very experienced in neural networks, it is important that the application is easy to use. Most of the technical details should be abstracted away so that only the most important and relevant information is displayed at any time. It will also be implemented as a web application to allow easy of use and avoid the need of installation.

2. **Expressive explanations** The explanations that are given by this tool also should be expressive as possible. The meaning here is that these explanations should target exactly why an image is classified or not classified in a particular way. Therefore, the objective of this project is for explanations to be as useful and detailed for users as possible.

3. **Human interpretable explanations** On the other side, the explanations cannot be too detailed that the user cannot pinpoint the most. Which is why the algorithms that generate explanations in this application should focus on the higher-level features e.g. body parts of an animal rather than low-level features such as curves and single colour blobs. Although these lower level features may be more useful to a machine, they are not usually that interpretable to humans.

4. **Flexible to architecture and implementation** The goal of the application is that it can be used by a wide variety of researchers and developers using neural networks for vision.

The application itself should be usable for as many neural network architectures as possible. Therefore the algorithms used to generate the explanations must also be as general as possible. Fortunately, some are general enough that they work on any classifier and not just neural networks. The application should also be able to support multiple implementations of neural network models in libraries such as Google's TensorFlow [AAB+16] and Caffe [JSD+14].

1.3 Challenges

Since this area of neural network explanations is a recent and rapidly innovative field, this is a challenging project to undertake with a large risk for failure. Therefore, there were a few challenges that we expected when undertaking this project:

1. **Performance of algorithms and application** Since neural network models can be huge with millions of parameters and images themselves have tens of thousands of pixels so optimising the performance of algorithms to relatively short time-frames (1-2 minutes) is crucial for this application to be usable. In the literature, there is a clear trade-off between processing speed and detail of explanations so the challenge in this project is to develop algorithms that have as interpretable explanations as possible while at the same time do not leave the user waiting for several hours for a result.

2. **Human navigability** As mentioned in the previous point, neural networks are very large and complex, therefore there is a lot of information that could be presented to the user. However, too much information can be overwhelming to the user so they cannot make any use of the explanations. The challenge for the application is to find a balance between detailed explanations and not overloading the user with irrelevant or useless information.

3. **Human interpretability** For images, there are tens of thousands of input features (pixels). An explanation could display the relative contributions of every single pixel of image and how it contributes to predicting the particular class. This explanation would be very difficult for a human to interpret as there is a cognitive limit to the amount of information that a person can take in at a time. Therefore the challenge is a compromise between how complex the explanations are and how interpretable they are to humans.

1.4 Contributions

There are three main contributions that we have produced for this project:

1. **Feature occlusion tool** One way of exploring features in networks for image recognition is finding regions of the image that contribute positively or negatively to a particular class. It is also interesting to see which regions are sufficient and which are necessary to classify an image. This is a tool does this by identifying regions and allowing users to occlude them so they can identify important regions. Its implementation is discussed in Chapter 3.

2. **Feature synthesis tool** Another way to explore high-level features that a neural network is to visualise the range of images that the network recognises as that particular feature. For example, in the case of classifying an image as a dog, its ears would be an identifiable feature. However, there are a variety of different shapes and colours of dogs ears. The aim of this tool is to synthesis a wide range of images that would be detected as dogs' ears in this case. This is a separate tool in the feature exploration application working alongside the region exploration tool. This tool is fully covered in Chapter 4.

3. **Feature exploration application** To make feature exploration and explanations of decisions more accessible to developers and researchers, a user-friendly application has been developed. It allows users to easily explore high level features in a neural network of their choice with particular image examples. It is extensible to different neural network architectures and implementations. It also easily extensible to other tools. This is discussed in Chapter 5.

Chapter 2

Background

In this section, we discuss the current state of the art of the fields that this project is based upon. Since the area of deep learning is so broad, we will only pay attention to a specific type of architecture called convolutional neural networks (CNNs).

In order to set the scene for those unfamiliar with the field, we will briefly discuss artificial neural networks, their applications. Then we will dive into a specific type of deep neural network called a convolutional neural network. We will also briefly discuss two of the most common implementation libraries for these networks: Caffe [JSD+14] and TensorFlow[AAB+16].

We will then go briefly into adversarial examples for neural networks. These examples, like the leopard print sofa considered in the introduction section, either predict the wrong class with high probability [MDFF16, DSZB17] or predict a class with a very high probability while not being identifiable to humans [NYC15].

Next, we will consider the current state of the art when it comes to generating explanations. We will briefly discuss what are known as "model explanations" i.e. explanations for the entire neural network. However, the main focus of this project are "prediction explanations" as

Finally, we will consider image synthesis techniques that will be built upon in the feature synthesis tool. In particular, we consider those that use optimisation that maximise activations of particular neurons but still approximating natural images. That way, we can see the range of images that the network recognises as a particular feature (e.g. a cat's nose).

2.1 Artificial Neural Networks

One can view artificial neural networks as models of the neural networks found in the human brain. These are computational models that are quite different from traditional sequential computer programs.

These are made up of small units of computation that are known as "neurons" and are a simplified version of the neurons in the human brain. Each of these neurons applies an affine function on a given number of inputs by multiplying each input by a fixed weight, summing the results and adding a bias term. A nonlinear activation function is then applied to the result in order to make the system more discriminative and introduce nonlinearity. A diagram representing a typical neuron is shown below in Figure 2.1. On their own neurons cannot even represent the XOR function since it is not linearly separable. However, the great power comes from connecting multiple neurons together to form a neural network. Generally neural networks are used to approximate functions. Though the function itself represented by $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ is not know, a subset of the domain X and its outputs. This *training data* is then used to help the network learn. It does this by altering its weights and biases until it can satisfy as much of the training data as possible.

There are many different ways to arrange the neurons in a network. One way that will not be discussed further is the recurrent neural network where a subset of the output is then used as

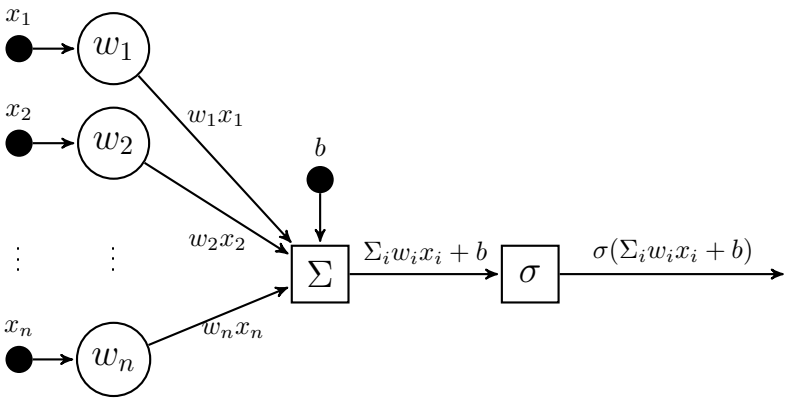


Figure 2.1: A diagram of a neuron. Each x_i is an input and w_i is the corresponding weight. The value b is a bias term and σ is the given activation function.

part of the input. These networks are usually used for tasks that have state and rely on previous information e.g. video processing. The other type are known as feedforward networks.

Feedforward networks get their name from the fact that data flows from inputs to outputs without any cycles. Since there are no cycles, there is no sense of "memory" meaning that they are only really suited for tasks requiring a single input and not sequential data. The main building block for these networks is the *layer*. Layers are collections of neurons where there are no interconnections between neurons in the same layer. Inputs to neurons in a particular layer come from neurons in the previous layer other than the *input layer* whose input is the direct input data e.g. an image or a collection of values. In a similar way outputs from neurons of a particular layer go to neurons in the next layer. Layers where inputs and outputs come from and go to all neurons in the previous and successive layers are known as *fully-connected layers*. The layers that are in between the input and output layers are known as *hidden layers* as they do not directly interact with the outside world. An illustration of a typical feedforward neural network is shown in Figure 2.2. What is interesting about feedforward neural networks is that they can approximate any

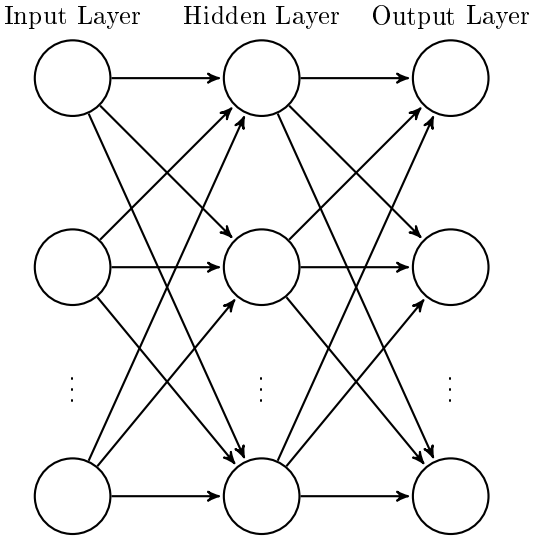


Figure 2.2: A diagram representing the typical feedforward network with a single hidden layer. Notice that data flows from the left-hand side to the right-hand side. These are also fully-connected layers.

function to arbitrary accuracy [Nie15]. In fact, with enough neurons they can even approximate any function with just a single hidden layer. The reason why multiple hidden layers, so-called *deep learning*, are used more often is because of the hierarchical structure of many inputs. For example, images are made up of so-called higher level features e.g. shapes such as triangles or textures.

These are further decomposed into lower. Deeper neural networks typically learn these hierarchical structures easier than shallower networks. In fact, it's been shown that deep neural networks will actually learn more complex features in deeper layers of the network [Le13, ZF14, YCN+15, Des16]

Feedforward networks are trained with training data using a method called backpropagation. It does this by minimising a loss function over the training data using gradient descent.A typical loss function for classification tasks is the mean square error (MSE) [Nie15]:

$$C = \frac{1}{N} \sum_{x \in D} \|y(x) - a^L(x)\|^2$$

where N is the total number of examples, $y(x)$ is the actual output and $a^L(x)$ is the output vector of the output layer. Notice that $a^L(x)$ can be expressed in terms of all of the parameters (weights and biases) of all of the neurons in the network due to the feedforward nature of the network. Backpropagation is done by finding the partial derivatives of the cost function in terms of the weights and biases and then applying gradient descent to adjust these parameters. These partial derivatives are then calculated by propagating the error backwards through the layers. The algorithm is made up of five steps [Nie15]:

1. **Input x**
2. **Feedforward:** Compute the neurons in each layer from the first to last
3. **Compute output error** δ^L
4. **Backpropagate error for each layer**
5. **Adjust weights and biases with partial derivatives**

This repeated over all of the training data and iterated many times (epochs). The calculations of partial derivative are summarised with the following equations [Nie15]:

$$\begin{aligned}
\delta^L &= \nabla_a C \odot \sigma'(z^L) \\
\delta^l &= ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \\
\frac{\partial C}{\partial b^l_j} &= \delta^l_j \\
\frac{\partial C}{\partial w^l_{jk}} &= a^{l-1}_k \delta^l_j
\end{aligned}$$

where σ is the activation function, a are the activations and z^l are the outputs of layer l before applying the activation function. The weight w^l_{jk} is the weight from neuron k in layer $l-1$ to neuron j in layer l.

A specific form of feedforward neural network called a convolutional neural network (CNN) has had many recent successes in image recognition and segmentation tasks [KSH12, SLJ+15, BHC15, LSD15], and is discussed in the following section.

2.1.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of feedforward neural network that have been particularly successful with image processing tasks. Pioneered by Yann LeCun in the late 90's with the LeNet-5 for digit recognition,they recently rose to prominence in recent years due to their success in the ImageNet Large-Scale Visual Recognition Challenge[RDS+15]) (ILSVRC) [SZ15, SLJ+15, KSH12, HZRS16]. The basic idea behind convolutional neural networks is that machine learning is extended to the feature extractors [LBBH98]. In the traditional model of pattern recognition, a hand-engineered feature extractor gathers relevant information from the image and eliminates variabilities that are irrelevant. A classifier is then trained to classify the resulting feature vectors. Convolution neural networks work on the same principle but the feature extractors are trained together with the classifier. In this section, we will discuss why CNNs have been so powerful in visual tasks as well as the main components.

Problems with Fully-Connected Networks

As mentioned previously, one way to recognise hierarchical features better is with more layers. However, for tasks related to analysing images, there will be many tens of thousands of connections between neurons. Even for the MNIST dataset with images of size 28x28 and so 784 pixels, if the first hidden layer of 100 neurons is fully connected then it would have nearly 80,000 connections. As well as increasing the time to train the neural network and memory resources it also increases the "capacity" [LBBH98] of the network. This means in order to avoid overfitting i.e. failure to generalise to unseen data, then an even larger dataset is required.

The other problem with a fully-connected neural network is that there is no built-in invariance with respect to translation. As LeCun argues in [LBBH98], typically images are centered and size-normalised to fit the input size of the network. This leads to variations in the positions of features in the image. A large enough network with data augmentation could learn to deal with these variations. However, there would be multiple units with similar weight patterns throughout the network as they would be recognising the same features but just at different positions in the image. Convolutional neural networks networks deal with this by replicating the same weight patterns (also known as *filters* or *kernels* across the space of the image. This also reduces the number of distinct connections so addresses the problem of capacity.

Another disadvantage of fully-connected neural networks is that they do not take advantage of the inherent local structure of 2D images [LBBH98]. It does not matter what fixed order the variables of the input are presented to a fully-connected neural network. The local correlations in images are why feature extractors work so well in pattern recognition tasks.

Convolutional neural networks combine *local receptive fields*, *spatial-subsampling* and *weight duplication* in order to ensure shift, scale and distortion invariance. A typical example of a CNN is the LeNet-5 [LBBH98] shown in Figure 2.3:

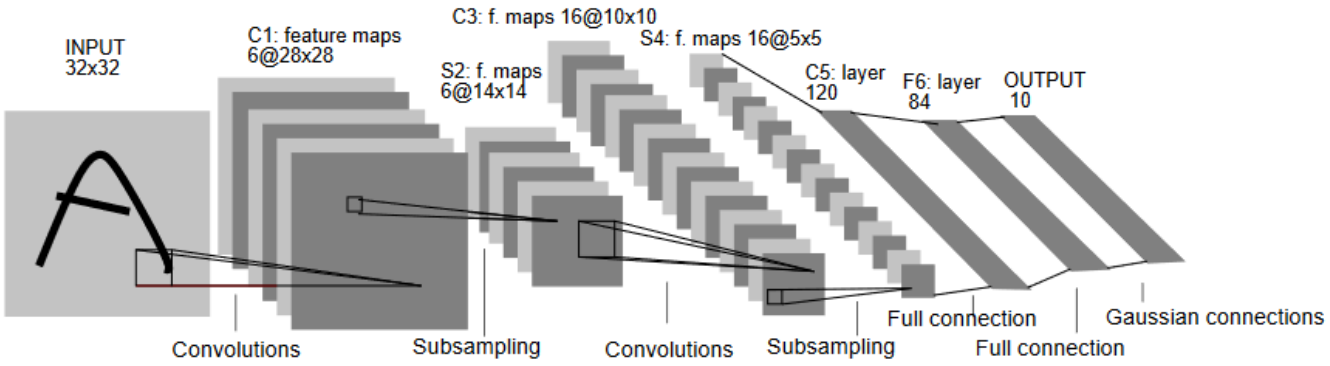


Figure 2.3: Architecture of the LeNet-5 convolutional neural network [LBBH98]. Each plane shown is a feature map.

Convolution Filters

Local receptive fields are called that because in convolutional layers (like C1, C3 and C5 in LeNet-5) each output neuron have a local neighbourhood of neurons in the previous layer. For the first layer (which is always a convolutional layer), these input neurons are pixels of the image. This means that only a small subset of the connections in the equivalent fully-connected layer are considered. For a particular *feature map* (output of a convolutional layer), the weights applied to every receptive field are the same which is also called a *filter* or *kernel* [Des16]. Filters work by moving over the image (convolving) and calculating a dot-product of the kernel with the input neurons (pixels in the case of the first layer). Like a regular feed forward layer, a nonlinear activation is applied. This is typically the *rectified linear unit* which is $f(x) = \max(x, 0)$. As an example in Figure 2.4, an input image of 32x32x3 (3 colour channels) with a 5x5x3 filter will produce a 28x28x1 activation

map. When an image is shifted, the resulting feature map is shifted the same amount. However, the exact position of a feature is less important than its relative position to other features. [LBBH98]. Each convolutional layer has multiple filters where each filter learns a particular feature. Exam-

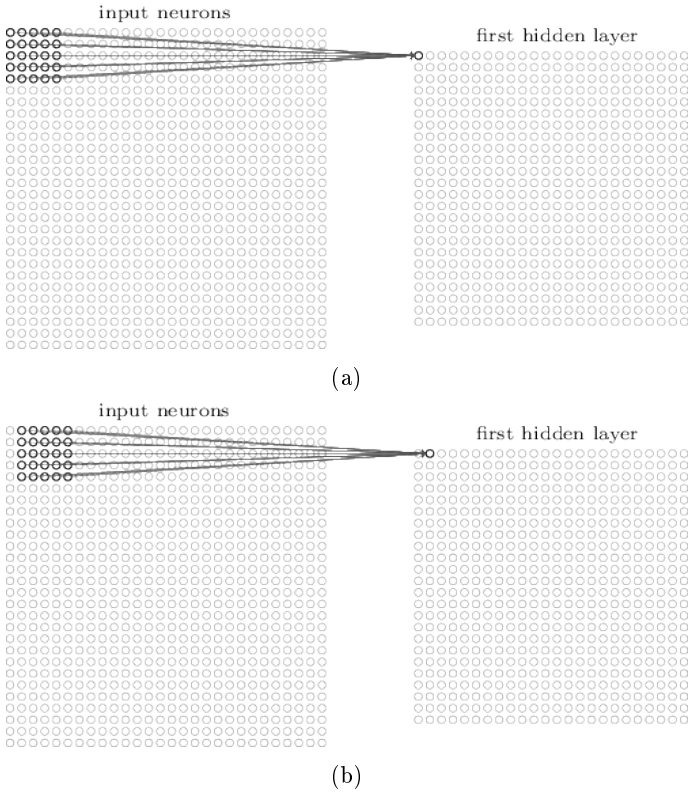


Figure 2.4: Illustration of a 5x5x3 filter applied to a 32x32x3 image (3rd dimension not shown) with stride 1 to get the resulting 28x28 activation map. In particular it shows the first (a) and second (b) neurons being computed. Adapted from [Nie15].

ples for first layer filters are shown in Figure 2.5. What is also interesting as one goes deeper into convolutional layers, more and more complex features are learned [ZF14, YCN+15, Des16]. Multiple convolutional layers are combined along with subsampling techniques such as pooling to form the feature extractor part of the network. A convolutional neural network typically ends with a series of fully-connected layers in order to classify the resulting feature vectors. These can also be represented as convolutional layers and some architectures semantic segmentation do not use them [LSD15, BHC15, SDBR14].

Pooling Layers

The spatial subsampling comes from special layers that are put between convolutional layers called pooling layers. These work by taking a neighbourhood of neurons in feature maps (typically 2x2 in size) and applying an aggregation function. Typical aggregation functions are the mean and maximum functions. These subsampling layers reduce the resolution and also the sensitivity of the output to shifts and distortions. These pooling layers can also be thought of as 2x2 filters with a stride of 2 and a norm replacing the normal activation function [SDBR14].

Training

Convolutional neural networks are trained in a similar way to normal feedforward networks by using backpropagation. They are also quicker to train with less training examples compared to the equivalent fully-connected network. This is because, as discussed before, there are substantially few connections due to local receptive fields and weight sharing.

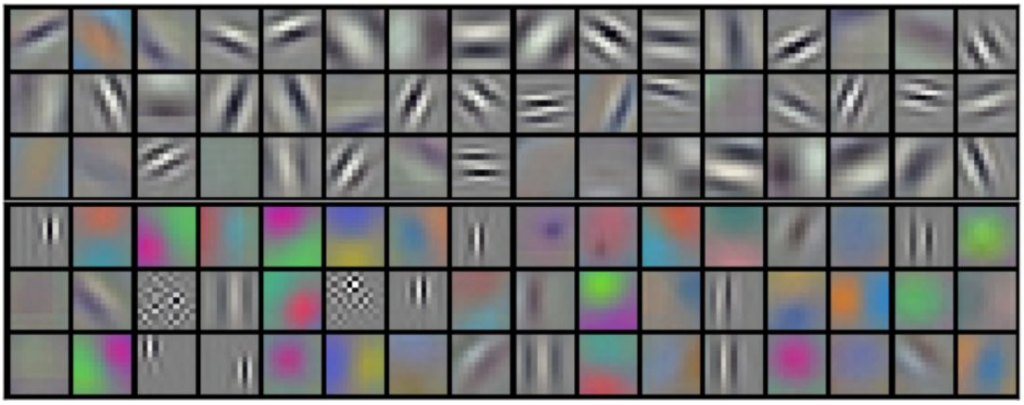


Figure 2.5: 96 kernels of size 11x11x3 learned by AlexNet [KSH12]. Notice that these are low-level features with edges in various directions and coloured blobs.

Applications

Convolutional neural networks have a wide range of applications although their main strength comes with visual applications due to the properties discussed as the beginning of this section. In particular, the best performing neural networks in the ImageNet[DDS+09] Large-Scale Visual Recognition Challenge[RDS+15] (ILSVRC) in recent years have been convolutional neural networks. The most well known task of this challenge being the image classification task where given an image, the network has to list all the classes present in image out of a possible 1000 classes that changes every year. There are also two other tasks that build upon this. The single-object localisation task presents these classes but also a bounding box for a single instance of each category. The object detection task extends this to *every* instance of each object category. Convolutional neural networks have been very successful in these tasks [SZ15, HZRS16, KSH12, SLJ+15].

Another task that convolutional neural networks have been successful at is semantic segmentation [LSD15, BHC15, HAGM14]. This involves taking an image and finding related regions corresponding to particular object classes. Four examples using two different networks, the fully convolutional network [LSD15] and that from [HAGM14], are shown in Figure 2.6. As can be seen, these networks are quite competent at detecting different classes even when they intersect (such as motorcyclists and motorcycles) and can even detect objects behind others such as the third example.

Finally, since convolutional neural networks are so well-suited to computer vision tasks they can also serve as components in architectures for more complicated tasks. An example would be image captioning. The task here is to produce a semantic description for a given image. Usually the convolutional neural network is used to produce an intermediate representation of the image with a recurrent neural network to generate the description [KFF15, VTBE15, DAHG+15].

2.2 Implementation Libraries

Writing one's own library for deep neural networks are prone to bugs and is time consuming. Since the main focus of this project is to explain how neural networks reach their decisions rather than training them, an implementation library for neural networks will be used for this project. In this section we will briefly discuss two of the most popular libraries for deep neural networks: Caffe[JSD+14] and TensorFlow[AAB+16]. Since neural networks will not be trained, the main criteria for using these libraries are easy-of-use and fast execution in deployment.

2.2.1 Caffe

Caffe is an open source C++ library developed by the Berkeley Vision and Learning Center (BVLC) for deep learning tasks. It is used to train and deploy convolutional neural networks but can also be used for other machine learning models. It supports GPU computation with CUDA and is also

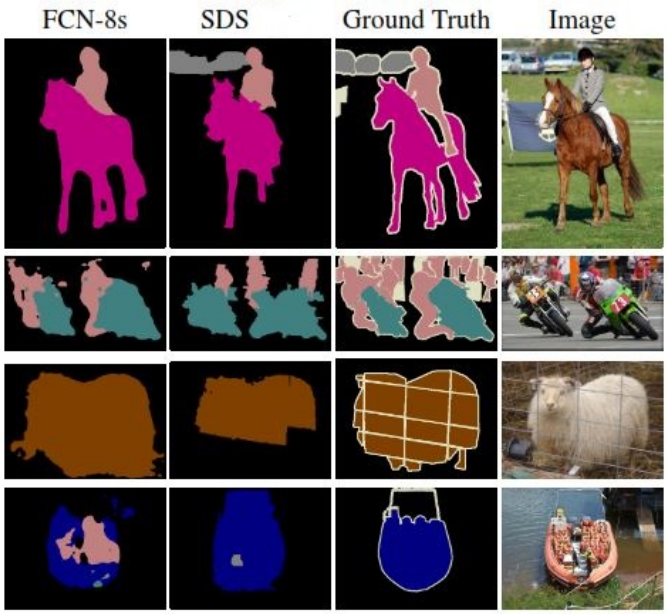


Figure 2.6: Four examples of images for segmentation with outputs from both [LSD15](first column) and [HAGM14](second column). (Adapted from Figure 6 of [LSD15])

Figure 2.7: Four examples of images and generated descriptions using an architecture that incorporates a CNN and an RNN. (Figure 6 of [KFF15])

efficient enough to process 40 million images per day on a single Titan GPU [JSD+14].

One of the interesting features of Caffe is that the representation of the model is completely separated from its implementation. This allows switching of models and experimentation as well as easy switching of deployment environments as well. These representations are in separate files in a JSON-like format so they are intuitive to understand. The library is also modular enough so that it can easily be extended to new data formats, network layers and loss functions although Caffe supports a wide variety of network layers already. Due to these representations, caffe can reserve exactly the right amount of memory needed for execution upon instantiation.

These network architectures are themselves represented as directed acyclic graphs. An example of an MNIST classifier architecture is shown in Figure 2.8. Data is communicated in arbitrarily-sized 4 dimensional arrays called blobs and layers have one or more input and output blobs. Blobs how wide varieties of data such as images, activations, parameters and parameter updates.

Although written in C++ so it is very efficient, there also exists Python bindings. This is very useful for this project as, in personal experience, Python has been shown to be ideal for rapid prototyping. It is also used by a wide variety of researchers in machine learning as well as so there is larger support from documentation and the community than most other languages. Caffe also provides some supporting tools to help developers debug their networks. An example would be the draw_net function, which allows the developer to visualise the graph of a particular network

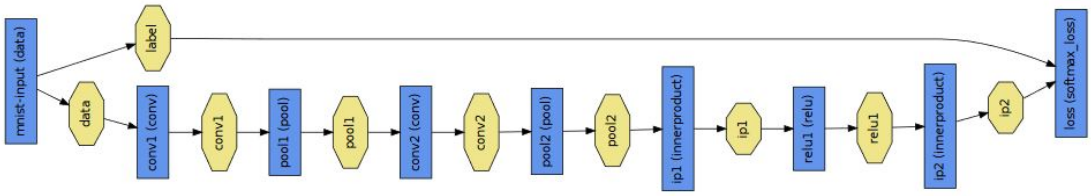


Figure 2.8: An example of a Caffe network architecture that classifies MNIST data. The blue rectangles are network layers while the yellow octagons are data blobs. (Figure 1 of [JSD+14])

by its prototxt file.

In terms of Caffe, there are also plenty of pre-trained models from the model zoo (https://github.com/BVLC/caffe/wiki/Model-Zoo) including popular models like AlexNet[KSH12] and VGG-16 and VGG-19 networks [SZ15]. As well as this, most of the papers we encountered for generating explanations and images had implementations based on Caffe. For these reasons and since the focus of this project meant that off-the-shelf neural networks were essential, we decided to use Caffe as the main implementation library.

2.2.2 TensorFlow

TensorFlow is an open source library written mostly in C++ and Python developed by Google [AAB+16]. It was built as a second generation system to DistBelief. It is used for dataflow programming in a wide variety of fields, but especially for machine learning tasks and deep neural networks. It is much lower level compared to libraries like Caffe and Keras which have individual layer constructs while TensorFlow is focused on operations.

Computations that are expressed in Tensorflow can be executed with little or no change on a wide variety of systems from mobile devices to large distributed systems. This is unlike a lot of libraries such as Caffe which can only be executed on a single device without extending the codebase [JSD+14]. Similar to Caffe, the representation is separated as much as possible from the implementation so that TensorFlow models can be implemented on a variety of hardware platforms. This reduces maintenance burdens as one does not need to have separate models for training and deployment as the same model can be run on different systems.

Like in Caffe, TensorFlow computations are expressed in the form of stateful dataflow graphs. An example of one representing a computation for neurons can be seen in Figure 2.9. There are two major differences that can be seen compared to the Caffe computation graph. The first is that the main computations are at a lower level compared to the layers that are used in Caffe. The second difference is that "blobs" are not used to pass data but tensors. Unlike blobs these are arrays of arbitrary dimension instead of four. Operations also can have parameters and control dependencies allow the developer to determine which nodes must be executed before others. These features allow developers to be much more flexible in their computations compared to more higher level libraries like Caffe, but that is not necessarily preferable for this project. TensorFlow also allows one to easily carry out machine learning tasks like training because it supports automatic gradient calculation. However, this complicates optimisation, particularly when it comes to memory usage. Since most gradient calculation methods require backpropagation, it means that tensors at the beginning of computation graphs have to be kept until the very end of the gradient computations.

Like Caffe, TensorFlow provides an API for Python but also in other languages such as Java, Go and Rust. As mentioned previously, Python is an ideal language for this kind of project due to its use in rapid prototyping. TensorFlow also provides tools like TensorBoard to visualise graph structures and statistics as well as EEG to visualise fine-grain information pertaining to performance.

Despite all these advantages, Caffe was chosen over TensorFlow due to two main reasons. The

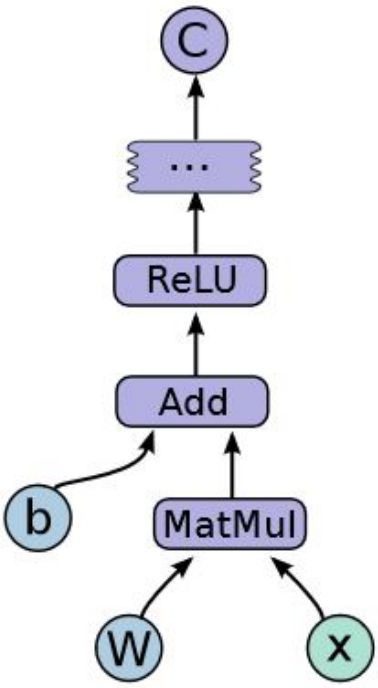


Figure 2.9: An example of a TensorFlow computation graph. The purple rectangles are computations while the blue circles are inputs. (Figure 2 of [AAB+16])

first is that Caffe as mentioned before is higher level compared to TensorFlow so it is much easier to express existing neural networks. Since we are not developing new architectures but explaining the decisions of existing ones, lower level expressiveness was not pertinent to this project. The second reason was that Caffe already has a lot of implementations of existing neural networks in its model zoo, making it very easy to rapidly prototype explanation algorithms as the example networks already exist.

However, while Caffe will be used in the primary development of the explanation algorithms, one of the goals of this project is to make it as implementation agnostic as possible. The application will also be extended to not only support Caffe models but TensorFlow models as well as other libraries (such as Keras) in the future.

2.3 Adversarial Examples

Explaining why a neural network has reached a correct prediction is important and helps build trust. They also give insight into if the network is considering the right features and discriminative boundaries [SWM17]. However, case where it gets it wrong are also important. Incorrect examples can be used to analyse the weaknesses of the network and allow researchers to improve models much more efficiently rather than by trial-and-error [Wel17]. As mentioned in the first section, there are plenty of natural examples that a human can easily recognise where even networks trained on those examples fail [Khu15]. As well as these examples that are present in the dataset, there are also methods discovered that generate adversarial examples. Even without using explanation techniques discussed later, these examples have provided some insights into how neural networks generally classify data. The two types of adversarial examples considered here are those that appear identical to humans but are classed differently and those that are not necessarily recognisable to humans but are confidently classed by a neural network.

2.3.1 "Identical" Images Classified Differently

The first type of adversarial example considered (and arguably the more interesting) are those that are imperceptibly different to real examples for humans, but will be predicted by a neural network

into two different classes. A good example would be the two images of Figure 2.10. Although these images look identical, the one on the left is correctly classified by AlexNet [KSH12] as a dog while the one on the right is classified as "ostrich".

Figure 2.10: An image (left) and its adversarial example which is identified by a neural network as "ostrich" (right). (Adapted from Figure 5(b) of [SZS+13])

This example was generating by solving the following minimisation problem [SZS+13]:

$$\begin{aligned} & \arg\min_{x} \|r\|_2 \\ \text{subject to: } & f(x+r) = l \\ & x + r \in [0,1]^m \end{aligned}$$

In other words, finding the minimum perturbation r of the original image x such that the neural network classifies it as the new label l. The exact computation is usually a hard problem but it can be approximated [SZS+13].

What is interesting about this method of generating adversarial examples is that it has a number of generalisation properties:

- For all networks and data sets studied, indistinguishable adversarial examples could be generated.
- *Cross model generalisation*: A large proportion of adversarial examples also applied to networks trained with different hyperparameters.
- *Cross training generalisation*: A large proportion of adversarial examples generated applied to networks trained with disjoint training sets.

These all suggest that adversarial examples are a universal problem for neural networks. However, it has also been found that training networks with these examples have made them more robust. Though we do not quite know why adversarial examples come up it is possibly due to their relative rarity but also absolute abundance (like rational numbers) [SZS+13]. Even though they are rare, due to their abundance they present quite a security risk as specially crafted images indistinguishable to humans can be used by malicious actors [MDFF16].

Moosavi et al. [MDFF16] expand on this with DeepFool, which computes these adversarial examples much more efficiently than the method mentioned previously. They also propose a measure of robustness classifier $\hat{k}$:

$$\rho_{adv}(\hat{k}) = \mathbb{E}_{\boldsymbol{x}} \frac{\Delta(\boldsymbol{x}; \hat{k})}{\|\boldsymbol{x}\|_2}$$

where $\mathbb{E}_{\boldsymbol{x}}$ is the expectation over the data. Using DeepFool to generate adversarial examples leads to an increase in robustness of networks trained on the MNIST dataset of around 50% for just a single epoch.

What is interesting about these examples is that explanation methods on images are also quite robust to these perturbations [MGB17, SCD+16]. However, the other type of adversarial examples are definitely not as robust since they are visually different to the classes that they represent at any natural images.

2.3.2 Examples Unrecognisable to Humans

As seen previously, one can generate adversarial examples that look the same as other images but. It is also possible to generate images that may looking nothing recognisable to humans but are predicted by neural networks with over 99% confidence in a particular class. Various examples from [NYC15] are presented in Figure 2.11.

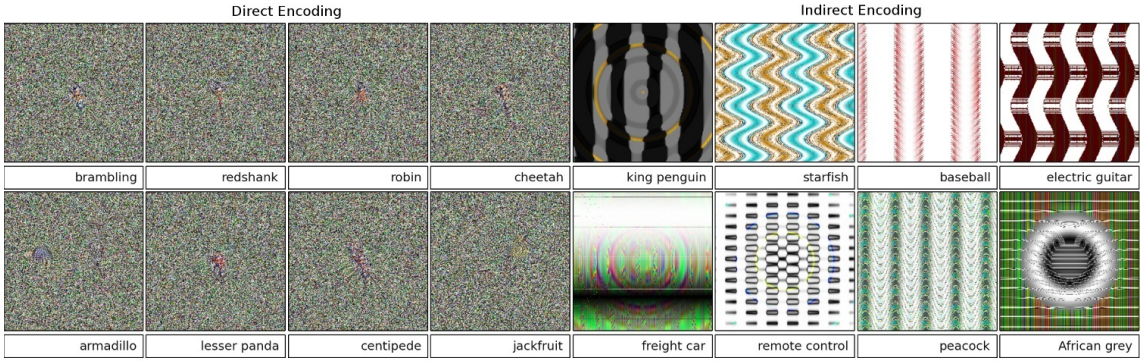


Figure 2.11: Adversarial images generated from the neural network AlexNet[KSH12] trained on ImageNet[DDS+09]. The left images show those generated directly from pixels while those on the right are from a compositional pattern-producing network (CPPN) [NYC15].

These images were generated using either evolutionary algorithms or gradient ascent that work well on both the MNIST and ImageNet[DDS+09] datasets. The indirect encodings tended to create more regular images (those on the right hand side of Figure 2.11) as the values of pixels tend to influence the values of other usually neighbouring pixels. For evolutionary algorithms, the fitness function was the highest classification confidence and the class with the highest confidence would become the new target class [NYC15].

For the MNIST DNNs, both directly and indirectly encoded images produced results that were human unrecognisable but with a confidence of over 99.99%. An interesting result is that for those indirectly encoded examples, certain patterns would appear in multiple versions for the same digit. For example those generated for the digit "1" had vertical bars. This suggests that the evolutionary algorithms exploit the discriminative features that DNNs learn [NYC15].

For the ImageNet DNNs, the direct encoding adversarial images are similar to those of MNIST DNNs although with colour as ImageNet images are in colour. What are much more interesting are the indirectly encoded examples. In this case, some of the patterns are quite obvious when the class it represents is known. For example, the image representing the remote control in Figure 2.11 has highly regular array of rectangles suggesting the buttons of the remote. The image representing the baseball has a red pattern on a white background, suggesting a baseball's stitching. The repetition of features also increased confidence which is in line with [SVZ13]. This means that deep neural networks do not tend to care about the overall structure of the image but the low and medium level discriminative features. The ImageNet DNNs could also be trained on these adversarial examples to be more robust to them: reducing median confidence from 88.1% to 11.7%.

The authors of [SVZ13] reason why high-confidence and unrecognisable images were produced by

genetic algorithms was because of the difference between of generative models and discriminative models. The DNNs used in the study were discriminative models meaning that decision boundaries are created between classes and they learn only $p(y|X)$. However, these boundaries could be much larger than the range of the training data for the corresponding class. This means one could make a very large perturbation of the image while still being in the same decision boundary. This is confirmed and investigated further in [GSS14] where these large decision boundaries are due to the locally linear nature and high-dimensional input of these models. Generative models, on the other hand, learn the joint distribution $p(y, X)$ so both $p(y|X)$ and $p(X)$. They are therefore much harder to fool as these images would have low marginal probability $p(X)$. Unfortunately, current generative models do not scale well to datasets like ImageNet due to the high-dimensional nature [SVZ13].

We have considered to the background of these models as well as interesting examples and implementation libraries. For the rest of this chapter we will discuss both explanation techniques as well as techniques to synthesise images from neural networks.

2.4 Explanation Techniques

Now with the background of neural networks and adversarial examples, we will consider explanation techniques that have already been looked at. Since this is such an important area in machine learning as discussed previously [Wel17, GF16], there have been a lot of research in various methods of explanation. It is important to define the difference between a *model explanation* and a *prediction explanation*. A model explanation attempts to explain the entire network which typically involves finding the explanations for each category or class when it comes to classifiers. A prediction explanation explains how a system came to the particular prediction for a given example. This project will consider prediction explanations primarily.

We will be considering visual explanations primarily such as heatmaps and regions of an image. This is first of all due to the fact that the models that we wish to generate explanations on are primarily focused on visual tasks. It therefore makes sense to use visual explanations, especially heatmaps that are overlayed on the original image. The second reason is that you can include much more information in a visual explanation without overwhelming the user unlike textual explanations, for example [VAMRL11].

This section will first go into general explanation techniques that can be applied to various systems, in particular classifiers. Next, we will consider various methods of providing visual explanations using points or heatmaps and comparing them with each other.

We will then look at a particular method of generating explanations that do not rely at all on the internal structure of the networks called *prediction difference analysis*. Finally, we will all discuss a couple of non-visual methods such as generating textual explanations from an image.

2.4.1 General Explanation Techniques

To begin with, we will consider general explanations that are typically model agnostic and can apply to a variety of classifiers and machine learning methods. These include techniques such as obtaining inequalities that explain a particular decision [KS17, Tur16] and feature selection [RSG16, VMGL12].

Inequalities and Regions

The first explanation method we will be looking at is the Model Explanation System (MES) from Turner [Tur16]. Although it provides model explanations, it does this through a composition of individual prediction explanations. A toy example from the paper is show in Figure 2.12. An example of an explanation is E_1 that explains that $f(x_1) = 1$ because $[x_1]_2 \leq 0.5$. This system is applicable to non-probabilistic black boxes that provide hard labels do so this would be applicable to neural networks. It also tries not to compromise accuracy for explainability which is very useful for our project.

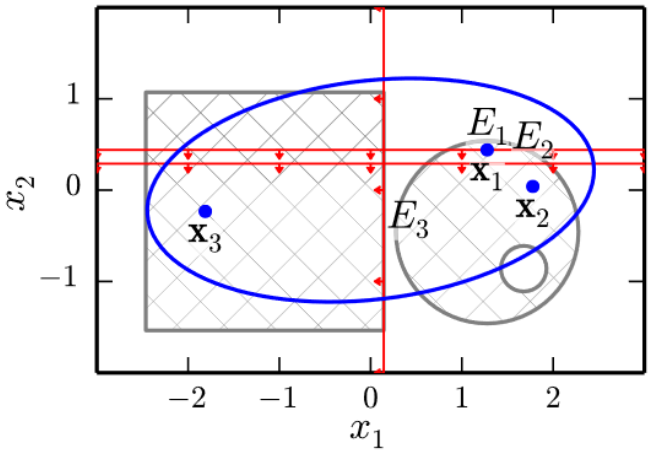


Figure 2.12: Example of the Model Explanation System on a toy classifier. The hatched regions are where f outputs 1 and 0 elsewhere. The test inputs are $\boldsymbol{x}_1, \boldsymbol{x}_2$ and $\boldsymbol{x}_3$ shown as blue dots. The red lines E_i are the respective explanations for each of test inputs. All the data comes from inside the ellipse so MES does not care about explanations failing at the plots extremities which each of these would. (Figure 1 of [Tur16])

.

The basic explanation for how model MES works is to find the explanation with the greatest quality score [Tur16]:

$$E^*(\boldsymbol{x}) = \arg\max_{E \in \mathcal{E}} S(E) \text{ s.t. } E(\boldsymbol{x}) = 1$$
$$S(E) = P(E(\boldsymbol{x'})|f(\boldsymbol{x'}) = 1) - P(E(\boldsymbol{x'})|f(\boldsymbol{x'}) = 0)$$

The score S is equivalent to the covariance and the null explanation has a value of 0 while the truth classifier has the score $S(f) = 1$. This means it will be chosen if it is in the set of possible explanations $\mathcal{E}$. This means one can tailor the possible explanations to those that are more human interpretable, though MES prioritises explainability over human interpretability. By construction, any explanation E that explains $f(x) = 1$ will not be selected for explaining why $f(x) = 0$.

Optimisation is done using Monte Carlo methods. The classifier in this case just needs to be queryable and that is also possible to obtain samples from the distribution $p(\boldsymbol{x})$. The set of possible explanations is expressed as a collection of explanation functions. In the extended version these are also parameterised:

$$\mathcal{E} = \{\mathbb{I}\{g(\boldsymbol{x}; \theta) \leq a\}, \forall a \in \mathbb{R}, \forall \theta\}$$

This extended version begins first by using optimisation to find the parameters for the explanation. Then the explanations are found using the original optimisation method to find the explanation that maximises the score $S(E)$.

This explanation can also be extended to image classification methods as well that highlight the regions of the image that contribute to the explanation. An example of this can be seen in Figure 2.13. While it shows relevant features of Powell's face, it does not show which are relatively more important which is explored in other methods later.

Another method extracts explanations through the use of a contribution matrix and non-negative matrix factorisation. A rule-like model is extracted from the model and in particular looks at model explanations as opposed to prediction explanations. It does these by finding a collection of constraints that explain a particular partition of the dataset, in this case class. A set of constraints

Figure 2.13: Example of an explanation in extended MES on an image of Colin Powell's face. **Left**: the original image. **Center**: the "explanation face". **Right**: the Hadamard product of the original image and its explanation face. Adapted from Figure 2 of [Tur16])

.

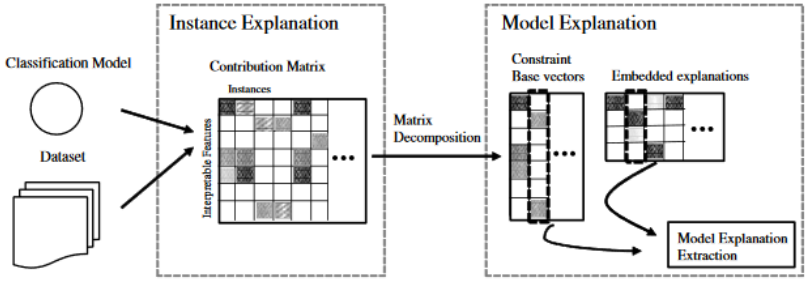


Figure 2.14: Overview of explanation extraction method in [KS17]

.

on features are called rectangles that are defined as follows:

$$B_{c,i} := \{(x_1, \ldots, x_2) \in \boldsymbol{S} : a_j < x_j \leq b_j, j = 1, \ldots, m\}$$
$$= \prod_{j=1}^{m} [a_j < x_j \leq b_j]$$

An explanation for a class is made up of the union of these rectangles for a particular class. The model explanation is then the collection of all of these class explanations. The quality of a model explanation is measured by its F1 score which is the average of the F1 scores for each of the class explanations.

The contribution matrix is found by finding the decision boundary of the interpretable model for every example in the dataset. If a contribution value is positive then that feature positively affects the decision of the particular example and the opposite for negative values. That means that the matrix is of size MxN where M is the number of features (e.g. pixels) and N is the number of data points. Obviously this will be quite impractical on the pixel level for image explanations. Not just due to the size of the matrix but also the fine-grained pixel level. If this was to be implemented for CNNs, it could be possible instead to use the higher level features learned by the neural network instead [Le13].

This contribution matrix is then turned into a non-negative one by separating the positive and negative matrices and then concatenating them to produce a 2MxN matrix. This matrix is then decomposed into matrices W and H such that WH approximates it as closely as possible. The rectangles are then constructed using the vectors in the W matrix and thresholds as w_l vectors are base vectors of the contribution matrix.

This method was found to have quite high performance with a minimal F1 of 0.72 and up to

0.94 [KS17]. However, the method itself produces model explanations which we are not too interested in and by the nature of the contribution matrix it would be quite computationally expensive for vision-related tasks unless higher-level features are extracted.

Feature Selection

A big reason why models like deep neural networks are so hard to interpret for humans is due to the high dimensionality of the data [VMGL12]. As mentioned previously, since images typically have tens of thousands of pixels, it is difficult to interpret which areas of the image were most relevant by looking at the features individually. One of the possible ways to deal with is through dimensionality reduction. It can be done in two ways:

1. Feature selection: Reducing the number of features by only keep those that are relevant
2. Feature extraction: Generating a smaller number of features based on the observed ones (e.g. PCA)

In this section we will be looking at feature selection using Local Interpretable Model-agnostic Explanations (LIME) [RSG16].

LIME works as an explanation technique for classifiers by finding a locally interpretable model around a particular prediction. It is also evaluated using not just simulated agents but also human experimenters. Not only could users select the better classifier but human non-experts could also determine methods to improve the classifier by identifying the irrelevant features from explanations. LIME uses methods that work both for text problems as well as image classification problems.

LIME tries to find the best explanation by finding explanation that is as faithful to the model as well as interpretable as possible. This is done by minimising both the loss compared to the original classifier in a local area ((L)) and the complexity of the explanation (Ω):

$$\xi(x) = \arg\min_{g \in G} \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

A toy example of LIME can be seen in Figure 2.15. Notice that the explanation (dashed-line) is only locally faithful. This is similar to the toy example in MES [Tur16] which is only concerned with the particular prediction.

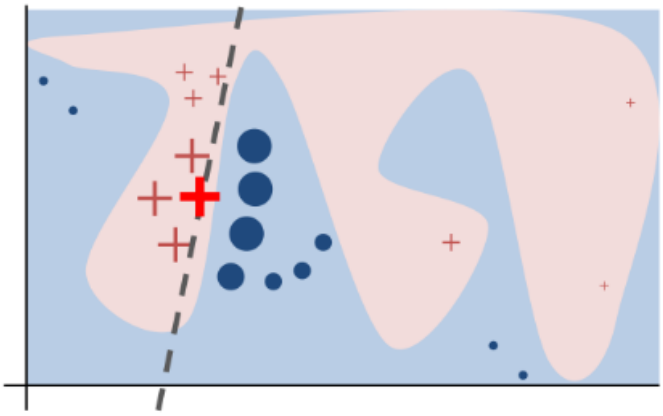


Figure 2.15: Toy example of LIME explanation (dashed line) for the prediction of the test input (red cross). (Figure 3 of [RSG16])

Spare Linear Explanations is the methods that LIME uses for individual predictions [RSG16]. It does this by finding the features that when used alone best predict the given class. The complexity of the explanation is defined by the number of features used up to a limit where it is infinite. For the text classification task, a bag of words is used while for image classification, a collection of super-pixels is used.

For explaining models, LIME does this by selecting a number of data samples based on their explanations using what the authors refer to as the submodular pick algorithm [RSG16]. Again, there is a hard limit of the number of an explanations that the algorithm will select. It does this by selecting the explanations so that as many features which are important in the explanations are selected as possible.

2.4.2 Visual Explanations

Now we will discuss various methods of explanation that are visual in nature. Most of these involve heatmaps and identifying areas of the image that are relevant to prediction. We begin by considering a various different methods used including CNN Fixations and gradient methods like sensitivity analysis. Then we will show that it was discovered that some of these methods are not theoretically correct even with a simple linear system [Kin].

Finally, we will look at the concept of distillation [HVD15] by converting complex models like neural networks into simpler but more interpretable models like decision trees.

Sensitivity Analysis

A rough way of achieving explanations for images (but not particularly accurate) is to use sensitivity analysis (SA) [SWM17]. Simply put, assuming that the classifier is described as function $f(\boldsymbol{x})$, then sensitivity analysis involves finding the partial derivatives of each feature:

$$R_i = \left| \frac{\partial}{\partial x_i} f(\boldsymbol{x}) \right|$$

In this case each feature is a pixel of the image. That means it looks at how changes in pixels affect the prediction. Sensitivity analysis does not explain $f(\boldsymbol{x})$ but instead looks at its variation. For example if an image of a rooster is occluded by flowers, the flowers are not themselves relevant but could be used to reconstruct the part of the rooster. Therefore it is not particularly useful for explanations.

Guided Backpropagation

A different approach is to look at particular neurons in the network and to see which regions of the image activate them the most. This is done by a gradient method known as guided backpropagation [SDBR14]. It is a modification of the deconvnet [ZF14] but gives more accurate images when reconstructing higher layers in the network.

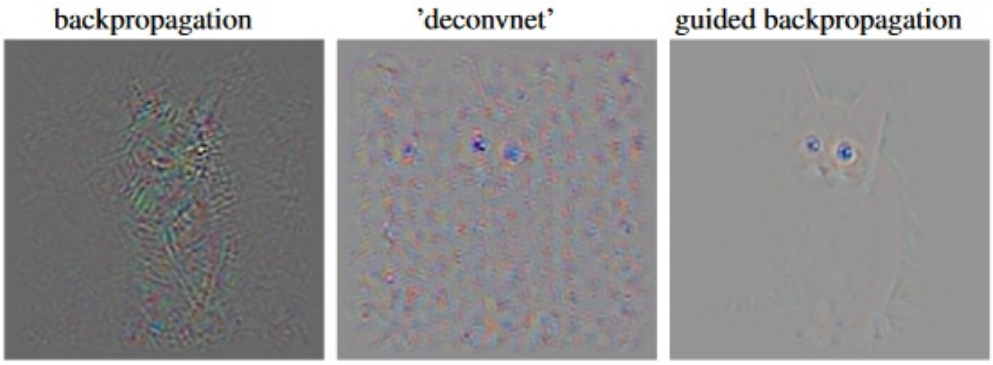


Figure 2.16: Comparison of guided backpropagation with regular backpropagation and deconvnet [ZF14]. (Adapted from Figure 5 of [SDBR14]).

This method involves calculating an estimate of the gradient through a backward pass of the network. Compared to deconvolution which just considers the top signal, guided backpropagation masks out values where either the top gradient (from backward pass) or the bottom activations are

negative for a particular layer. This is called guided backpropagation as an extra guidance signal is being given to regular backpropagation.

As can be seen in Figure 2.16, it is able to compromise between the accuracy of localisation of backpropagation and the clarity in parts of the "deconvnet". While it can clearly visualise important areas of image, it does not seem to be good at visualising which relevant regions are more important than others.

Grad-CAM

Another method of producing visual explanations is to produce a localisation map of the particular class. One method of doing this is through Gradient-weighted Class Activation Mapping (Grad-CAM) [SCD+16]. An example for the class "tiger cat" is seen in Figure 2.17. This is a generalisation of a method called Class Activation Mapping (CAM) [ZKL+16] which only works for certain convolutional architectures. Grad-CAM generalises to all forms of convolutional architectures. It works by using the gradients of the target class into the final convolutional layer in order to produce the coarse localisation map.

Figure 2.17: An example of an image and the Grad-CAM map for the class "tiger cat". Notice that although it is a coarse map it still localises well to the cat. (Adapted from Figure A8 of [SCD+16]).

This is calculated by first calculating the gradient of the score for the particular class c, y^c with respect to feature maps A^k of a convolutional layer. The gradients are global-average-pooled in order to find the neuron importance weights α_k^c:

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k}$$

In order to obtain Grad-CAM for the class, a weighted combination is performed with the forward activation maps and ReLU is then applied to remove negative values:

$$ReLU\left(\sum_k \alpha_k^c A^k\right)$$

This is map only corresponds to the size of the filters in the last convolutional layer (14x14 in the case of VGG[SZ15] and AlexNet[KSH12]) meaning that it is quite coarse. However, it still performs quite well. Grad-CAM visualisations are more faithful to the model compared to previous models for example [SCD+16]. This is because paths that changed the CNN score had higher intensity for Grad-CAM. Another observation was that at deeper layers, Grad-CAM became more accurate and less sparse. This is probably due to the higher level features that are obtained in deeper layers. One can find counterfactual explanations i.e. what regions would make the network change its decision by negating the elements when calculating the neural importance weights $alpha_k^c$.

Grad-CAM also allowed the authors to find a statistical bias in their dataset. They found that CNNs were misclassifying doctors and nurses as each other. It was found using Grad-CAM that the most relevant features of the image including the hairstyle which suggests that there was a bias towards female nurses in the dataset.

Although these visualisation methods are interesting, they still give quite coarse or inaccurate results. The next method we will consider went about it a different faster way without calculating gradients to produce more detailed explanations.

CNN Fixations

A method proposed in 2017 has found to give even more accurate results than previous results which involves finding CNN fixations. Essentially, these are discriminative image regions that are found by propagating information backwards [MGB17]. It provides a prediction explanation for a given input and class given although it can be extended to captioning networks as well. The

Figure 2.18: **Left**: An image correctly classified as "English Setter". **Centre**: The CNN fixations calculated for that class. **Right**: The heatmap calculated from the fixations.

algorithm essentially works by starting off with a single index for the class predicted. Then, by using the activations of the current layer and the weights from it to the deeper layer, relevant activations (those greater than 0). The indices of these neurons are then used for the next layer and so on until the indices are at the pixel length at the start of the network. The paper also provides algorithms for calculating the fixations in inception [SLJ+15] and residual [HZRS16] modules.

The advantage that this algorithm has over gradient methods is computational speed [MGB17]. Most of the existing methods were based on gradient calculations whereas CNN fixations can be calculated using just the results from the forward pass. CNN fixations are also typically more localised to the object compared gradient methods and so are more object. The CNN fixations are much higher resolution (pixel-wise) compared to Grad-CAM [SCD+16] (size of convolution filters), for example.

CNN fixations are in particular useful for explaining why incorrect predictions were made by a classifier by identifying the local area of the image that was considered. An example would be the first example in Figure 2.18 where it correctly identifies a loafer even though it fails to label the image correctly as a suit. CNN fixations were also found to be superior to existing gradient methods when it came to predicting saliency.

Layer-Wise Relevance Propagation

Another method of calculating the relevance of regions of an image for a specific prediction in neural networks is Layer-wise Relevance Propagation (LRP) []. It explains predictions relative to the state of maximum uncertainty and does this by redistributing the classifier prediction $f(\boldsymbol{x})$ backwards until the input variables are reached. Relevance is than assigned to each of the input variables (pixels) and so can be displayed as an image. The key property is that relevance is

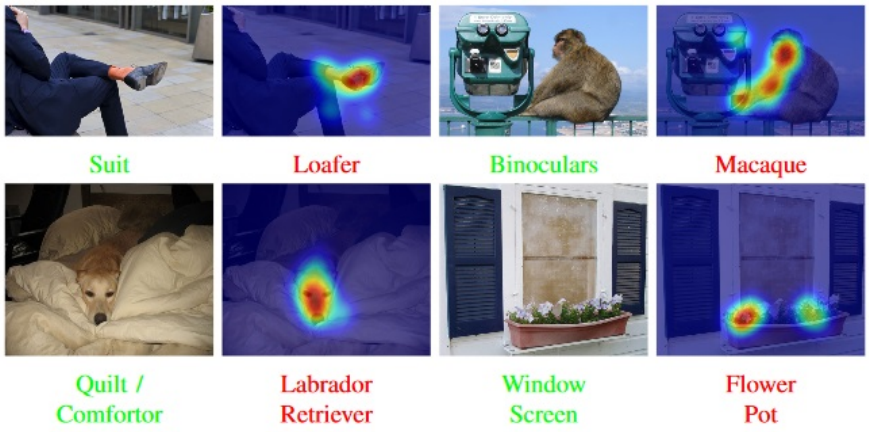


Figure 2.19: Explaining wrong recognition results from VGG[SZ15]. The green label is the actual label whereas the second image of each pair has the wrong label with the explanation (Figure 7 of [MGB17]).

conserved throughout the layers:

$$\sum_i R_i = \cdots = \sum_j R_j = \sum_j R_j = f(\boldsymbol{x})$$

The standard redistribution rule of relevance from one layer $l+1$ to a neuron in the previous layer l is as follows:

$$R_j = \sum_k \frac{x_j w_{jk}}{\sum_j x_j w_{jk} + \epsilon} R_k$$

where x_j are activations of layer l, R_k are relevance scores for neurons at layer $l+1$ and $w_j k$ is the weight from neuron j to neuron k. An alternative rule is to separate positive and negative activation and weight pairs and have difference coefficients α and β:

$$R_j = \sum_k \left(\alpha \cdot \frac{(x_j w_{jk})^+}{\sum_j (x_j w_{jk})^+} - \beta \cdot \frac{(x_j w_{jk})^-}{\sum_j (x_j w_{jk})^-} \right) R_k$$

where $\alpha - \beta = 1$. Compared to sensitivity analysis [SWM17], LRP truly composes $f(\boldsymbol{x})$. This method allows LRP to distinguish between positive and negative influences like Grad-CAM but includes then in the same map while deconvolution [ZF14] and CNN fixations [MGB17] do not show negative influences.

When evaluated using the method described in [SBM+17], LRP is both subjectively and objectively superior to SA. This is mainly due to the fact that SA does not indicate each pixel's contribution but measures the sensitivity of $f(\boldsymbol{x})$ to change. It is also much faster than methods that give positive and negative influences such as prediction difference analysis that will be discussed later as it only requires a single forward pass of the neural network.

Real-time Image Saliency

Another possible method to generate explanations for predictions on images is to generate a separate model that produces saliency masks [DG17]. Once trained, the model runs very quickly and in real-time with 100 masks computed per second. Saliency masks show which regions of the image are most relevant for the model for the particular input class. This can be obtained for example by finding the smallest region whose removal causes classification score to drop significantly. However, most algorithms are time consuming due to the nature of the optimisation. There are two main regions outlined in [DG17] that are important when it comes to deciding how a model made its prediction:

1. Smallest sufficient region (SSR) - smallest region of the image that alone allows confident classification

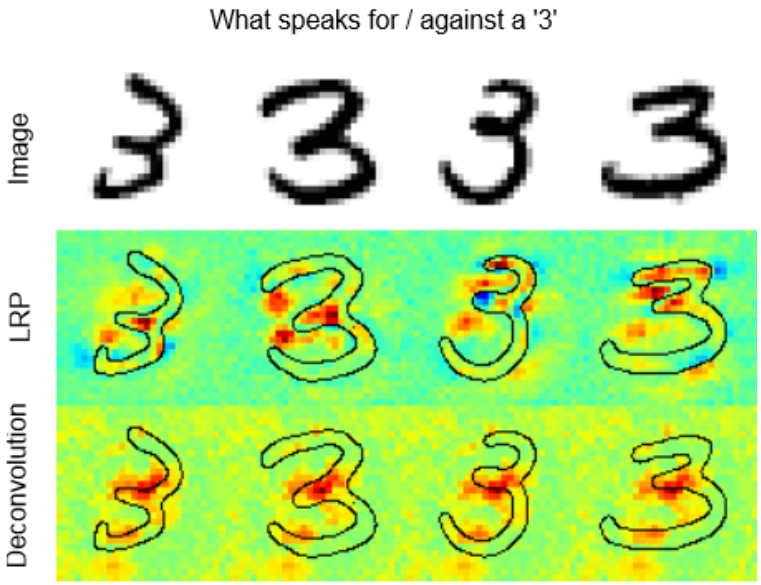


Figure 2.20: Comparing the explanations of LRP and deconvolution [ZF14] for four examples of the digit '3'. Note that LRP has both positive (red) and negative (blue) evidence. (Adapted from Figure 4 of [SWM17]).

2. Smallest destroying region (SDR) - smallest region that when removed, prevents a confident classification

It is best to find a region that performs well as both an SSR and an SDR to obtain informative explanations [DG17]. The problem is that most methods of removing evidence to obtain these kinds of regions also lead to more information being introduced so this new evidence must be minimised. An easy way to manipulate an image is with a mask M element-wise. However, adversarial artifacts can mean even a mask of a small magnitude can completely change the predicted class [MDFF16, DSZB17]. One can add an alternative image A in order to reduce these artifacts so the resulting explanation is:

$$E = X \odot M + A \odot (1 - M)$$

This alternative image can be blurred version of the input image X so that it is hard to generate high-frequency, high-evidence artifacts. A total variation (TV) penalty should also be added to encourage smoothness in the image.

The authors also outline the following saliency metric for probability of the chosen class p and a as the fraction of the total area of the image that the saliency region is:

$$s(a, p) = \log(\tilde{a}) - \log(p)$$

where $\tilde{a} = max(a, 0.05)$ to prevent instabilities that come with smaller regions. Low value of saliency metric indicates a good saliency detector as it tries to find as small a region as possible that still predicts the target class i.e. the SSR.

The mask can be found by optimising the objective function L:

$$L(M) = \lambda_1 TV(M) + \lambda_2 AV(M) - log(f_c(\Phi(X, M)) + \lambda_3 log(f_c(\Phi(X, 1 - M))^{\lambda_4}$$

The first term encourages smoothness while the second term encourages the smallest region possible. The third term encourages the correct class to still be recognised (SSR) while the fourth term ensures that removing the region leads to predicting a different class (SDR). Finding the region iteratively is slow and usually overfits so that the resulting mask is blurry and imprecise. Instead, a U-Net [] is used to produce the mask from an image. The original network trained on has its weights fixed for each layer has its activations fed into the masking model while the last layer also goes under a feature selector for the target class.

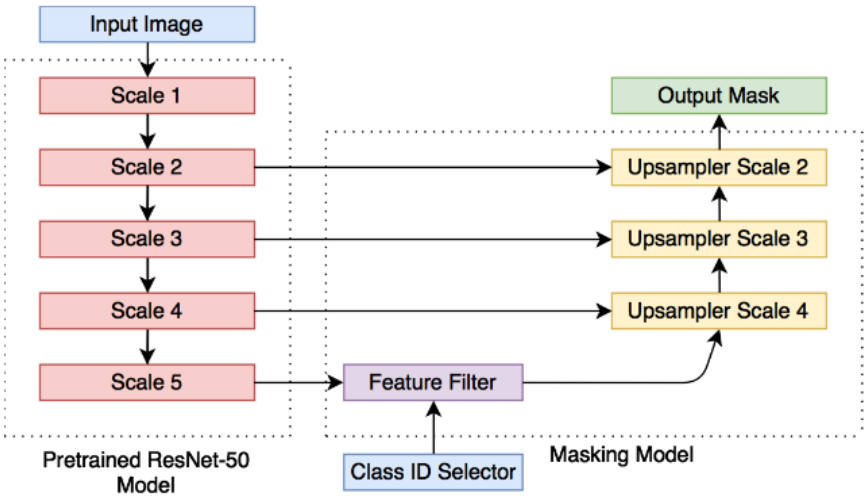


Figure 2.21: Architecture diagram of the masking model when trained on a Res-Net50 [HZRS16] architecture. (Figure 4 of [DG17]).

When compared to popular methods, the masking model trained with the GoogLeNet model performed to best at the weakly supervised object localisation task. This masking model also had the lowest value in the saliency metric of 0.318. So this is an accurate and efficient method for calculating a saliency map for a prediction once the masking model is trained. The only problem with implementing this is that the model training my take too long for the user.

PatternNet and PatternAttribution

A lot these methods such as DeConvNet [ZF14], Guided Backpropagation [SDBR14] and Layer-wise Relevance Propagation (LRP)[SWM17] produce interesting and some seemingly accurate results. However, none of these algorithms are actually theoretically correct when it comes to simple linear models [Kin]. Since neural networks are nonlinear anyway, the best one can do is approximate using these algorithms. However, the authors of [Kin] propose two algorithms that are theoretically correct for linear models and so produce better approximations for nonlinear models.

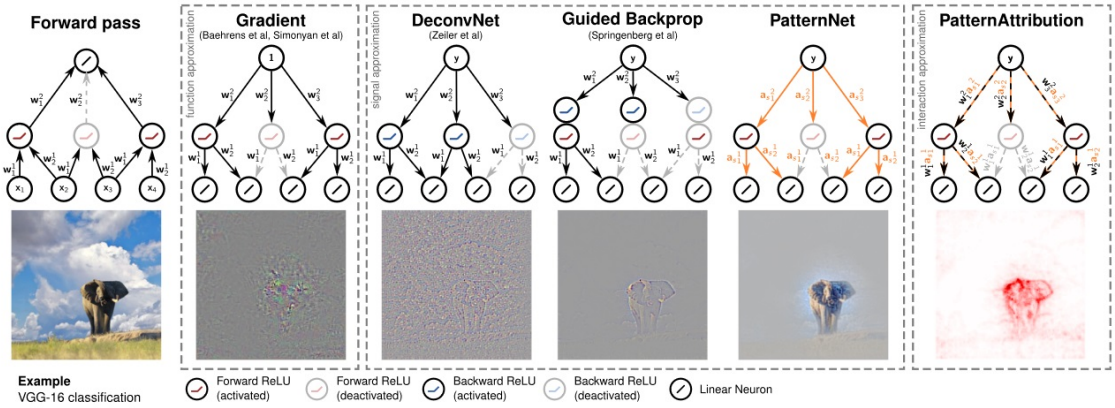


Figure 2.22: Various signal explanation methods compared to PatternNet. A visualisation of attributions using PatternAttribution is also given. (Figure 1 of [Kin]).

The author's have a toy example of a linear model where data samples are composed of a *signal* and a *distractor*. The goal of methods like DeConvNet and Guided Backpropagation is to obtain the signal that causes the network activations. To extract y which helps create the signal,

the weight vector or *filter* w has to be found that filters out the distractor. In general, the filter does not follow the direction of the signal. The reason why DeConvNet and Guided Backpropagation are not theoretically correct is because in the linear model, the visualisations produced show the filter w and so neither the signal nor the pattern to produce the signal.

For finding attributions like in LRP, the optimal attribution would be multiplying the signal by the weight vector w [Kin]. However LRP, like DeConvNet and Guided Backpropagation, does not extract the signal from the linear model so the attributions cannot be optimal.

What the authors of [Kin] propose is a method of obtaining the signal in linear models and hence a more accurate signal in deep neural networks. This is done by using a two component signal estimator for positive and negative values:

$$S_{a+-}(\boldsymbol{x}) = \begin{cases} a_+ w^T x, & \text{if } w^T x > 0 \\ a_- w^T x, & \text{otherwise} \end{cases}$$

Where a_+ and a_- are patterns that are calculated. PatternNet is a back-projection of this estimated signal to input space. Since it is a more accurate signal results, it produces more accurate results than both DeconvNet and Guided Backpropagation as can be seen in Figure 2.22. PatternAttribution also uses this more accurate signal estimate to produce better maps of relevancy. As well as being subjectively better, PatternNet and PatternAttribution are objectively better as the signal estimator had a greater score than for the other methods.

It seems that PatternAttribution would be the best way to produce a map of both positive and negative evidence for the region exploration tool. It is faster than methods like [DG17] as a model is not required to be trained but more accurate than LRP or RP.

Distilling to a Decision Tree

A completely different method for generating explanations is instead to convert the model itself to a more interpretable model such as a decision tree [FH17]. The normal trade-off of machine learning models is between performance and interpretability. Neural networks and especially those with many hidden layers achieve high accuracy as shown previously but are not interpretable at all to the point of being "black boxes" [SWM17]. On the other hand, decision trees have some of the lower performances of machine learning models but are very interpretable to humans due to their hierarchical structure. The aim of distillation is to transfer the knowledge learned by a cumbersome model to a simpler model [HVD15]. In this case, the resulting model takes advantage of the higher interpretability of decision trees to make the predictions more explainable.

Specifically, the resulting model is a soft decision tree where decisions are made under a probability distribution rather than deterministically. These probabilities are propagated down to leaf nodes that choose a distribution of class labels according to these probabilities. This decision tree is trained using a loss function that minimises the cross entropy between leaf nodes weighted by path probability and target distribution:

$$L(\boldsymbol{x}) = -\log\left(\sum_{l \in leaf\ \ nodes} P^l(\boldsymbol{x}) \sum_k T_k \log Q_k^l\right) T_k \log$$

where Q_k^l is the distribution of class labels associated with that leaf node, $P^l(\boldsymbol{x})$ is the probability of reaching the leaf node l and T is the target distribution. A regulariser term was also added encourage inner nodes to make more equal use of sub-trees. Leaving this out would mean inner nodes would plateau by assigning high probability to one subtree [FH17]. By the nature of decision trees, less data is considered at deeper nodes meaning the computation of probabilities becomes less accurate. Results were improved by having a decaying running average in order to counteract this.

As an experiment, a decision tree was trained from the true targets of the MNIST dataset and then a second one was trained from the soft targets of a pre-trained neural network. While the decision tree on its owned achieved a test accuracy of 94.45% while the one trained from the neural network achieved a higher test accuracy of 96.76%. The reason why this decision tree is much

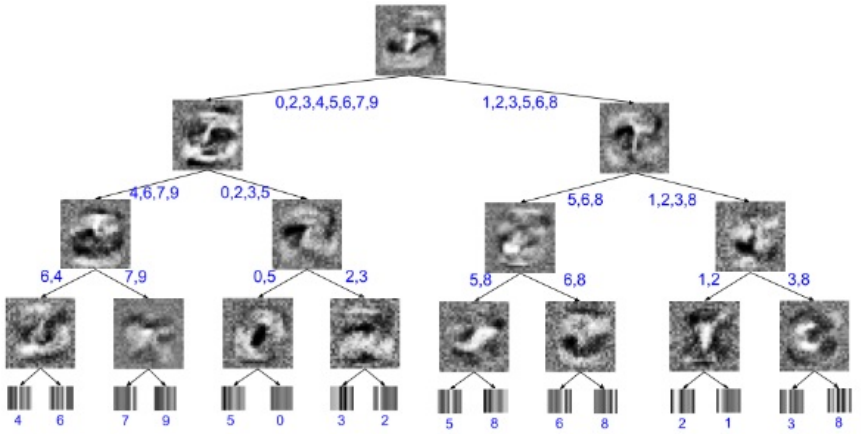


Figure 2.23: An example of a distilled decision tree of depth 4 from a neural network trained on the MNIST dataset. The images at the internal nodes are the filters learned by each node while those at the leaves are the probability distributions of the output classes. An interesting example is the one on the bottom right that looks for the closing part of the loop that would turn a prediction from a 3 to an 8. (Figure 2 of [FH17]).

more interpretable is that it turns the hierarchical representations that neural networks are good at into hierarchical decisions which are easier for humans to understand. Unfortunately the authors have not gone into distilling knowledge from networks trained on natural datasets like ImageNet [DDS+09]. Due to the large number of classes displaying the whole decision tree would not be practical but visualising the filters learned by nodes actually taken by an input should be more interpretable.

2.4.3 Prediction Difference Analysis

An interesting method of explanations for classifier is *prediction difference analysis* [ZCAW17]. The main idea behind it is to look at how the probability of predicting a particular class changes when features in the input are not considered. This section will consider the original paper on the method [RŠK08] as well as an extension more suited to images and neural networks [ZCAW17] and another that has a creative method of visualisation for neural networks [DSZB17].

Initial Method

This method was first introduced by Robnik and Kononenko in 2008 [RŠK08]. This works with any classifier that outputs probabilities os most deep neural networks. It uses the causal effect of changing an attribute's value in order to determine its relevance. This achieve by observing the "prediction difference" i.e. the change in probability for a particular class when an attribute's value is unknown. The higher prediction difference, then the more relevant that particular attribute is to predicting the class in the given instance.

The paper also lays out three different ways of calculating the prediction difference for an input x, prediction y and attribute i:

1. **Information difference**: $\text{infDiff}_i(y|x) = \log_2 p(y|x) - log_2 p(y|x\backslash A_i)$
2. **Weight of Evidence**: $\text{WE}_i(y|x) = \log_2 p(y|x) - log_2 p(y|x\backslash A_i)$
   $\text{odds}(z) = \frac{p(z)}{p(\overline{z})} = \frac{p(z)}{1-p(z)}$
3. **Direct probability difference**: $\text{probDiff}_i(y|x) = p(y|x) - p(y|x\backslash A_i)$

The last method is typically not used as it is difficult for humans to comprehend and evaluate probabilities, especially those that are close to 0 or 1.

Calculating $p(y|x)$ is usually fairly straightforward as most classifiers have some sort of probability distribution across classes as the result. Calculating $p(y|x\backslash A_i)$ is usually more difficult although it depends on the classifier. For Naive Bayes classifiers, one can just exclude that attribute for the input. For a neural network, one can set the weights outgoing from the neuron to 0. For nominal values in other classifiers, one can replace the attribute value with all the possible values weighted by each value's prior probability. For numerical values one can discretise the value space and apply a similar method to the intervals.

For these explanations, six interesting properties were found:

1. Model dependency: if model is wrong, explanations will reflect that
2. Instance dependency: different explanations for different instances (obviously)
3. Class dependency: different explanations for different classes and difference attributes have different influence on different classes
4. Capability to detect strong conditional dependencies
5. Inability to detect and correctly evaluate utility of attributes' values if a change of more than one attribute is needed to change predicted value (e.g. pixels in an image).
6. Dependency on type of difference evaluation

These explanations are then visualised with a system called explainVis that shows how the most relevant attributes contribute to the particular class. An example on the Titanic dataset can be seen in Figure 2.24. Explanations can be evaluated by calculating the euclidean difference between the prediction differences of attirbutes and the "true explanation".

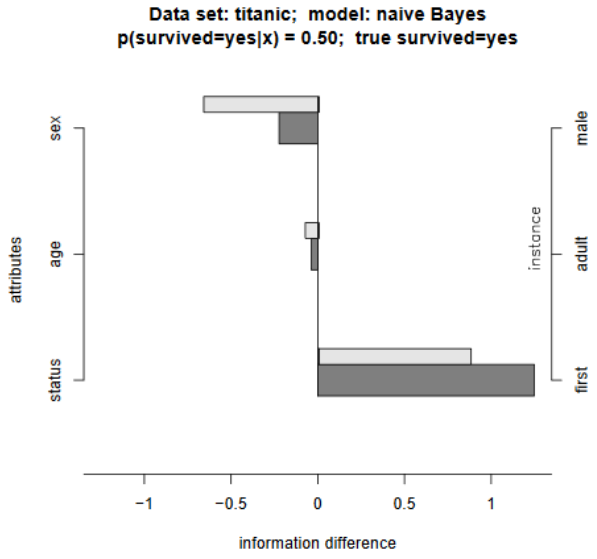


Figure 2.24: Example explanation of a first class adult male passenger from the Titanic dataset. Explanations for this instance are dark bars while the light bars are the average explanations for these attributes' values. (Adapted from Figure 2 of [RŠK08])

Extension to Images

The original method works well with a lot of classifiers and input data but not particularly well with images. The first reason why this is the case is because of the fifth property of the explanations. Rarely will removing a single pixel from an image cause a classifier to predict a different class. The second reason is that there are too many features (typically tens of thousands) to visualise in the method used by explainVis. Therefore Zintgraf et al. [ZCAW17, ZCW16] extended prediction

difference analysis to images instead.

The second issue was addressed simply by overlaying the prediction differences over the original test image and using colour to indicate positive (in red) and negative (in blue) influences. An example we analysed by reproducing the code is shown in Figure 2.25.

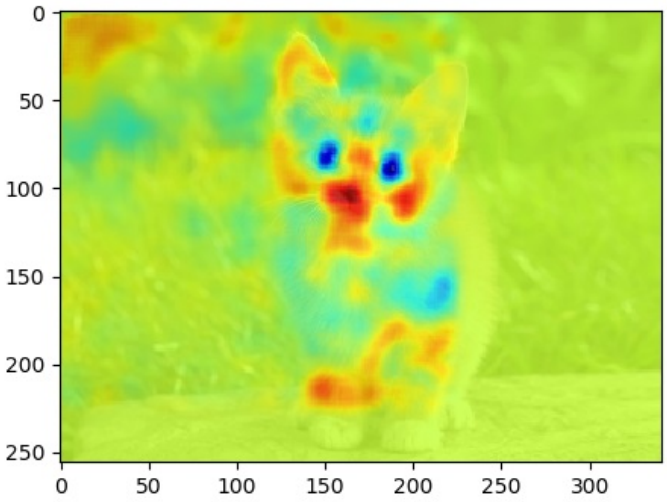


Figure 2.25: An example of the visualisation method in prediction difference analysis. Red are influences for the class "Egyptian cat" and blue are against. Its nose, cheeks and right ear are most supportive for the classification. Interestingly, the eyes constitute strongly as negative evidence against it.

Instead of a single pixel being made unknown, a whole patch of pixels is instead removed. It was found that a patch of 10x10 pixels gave a good trade-off between smoothness of the results and sharpness of the patches [ZCAW17]. A method to remove a particular attribute in neural networks would be to retrain without but this is impractical for these deep neural networks. Instead, the authors found that conditionally sampling on a slightly larger neighbourhood (14x14 pixels) approximated making the patch of pixels unknown sufficiently.

The authors also contributed a method for visualisation specific to neural networks. In this case, they visualised the feature maps of several hidden layers. For each feature map, the relevance is calculated for each neuron. For convolutional layers with multiple feature maps, these relevance vectors are average over all units in that feature map. What was also found by the authors was the difference in relevance between the penultimate and output layer. In the penultimate layer, for the top 3 scoring classes e.g. different subspecies of elephant, the visualisations were broadly very similar. However, in the output layer, these visualisations look quite different where the ears are the crucial difference.

What the authors also found was that different neural network architectures can sometimes find different pixels are more relevant for the same image and prediction. For example, when comparing the AlexNet[KSH12], [] and VGG network[SZ15] on an example of a hot air balloon. AlexNet would focus more on the contextual information such as the blue sky while the VGG network would instead focus purely on the basket. Since the VGG network's second highest prediction was a parachute, it identified the basket as the distinguishing feature between parachutes and hot air balloons.

A problem with prediction analysis for this particular project is that the user will usually expect reasonably high responsiveness even if they are machine learning experts who are used to long waiting times for training. Since the algorithm is dependent on the number of pixels, this may not be practical as a tool in the web application.

Different Visualisation for Neural Networks

A different method for visualising explanations in an interpretable way is presented by Dong et al. [DSZB17]. Though used to detect adversarial examples, it could also be used for pinpointing concepts detected in an image.

It works by performing prediction analysis on the neurons in the fully connected layers of the network and the neurons with the highest absolute values are selected. Then, since neurons typically identify higher level concepts in deeper layers [Le13, ZF14, YCN+15, Des16], the images that provide the highest activations are found and displayed. A few examples of this, with real and adversarial images, are shown in Figure 2.26.

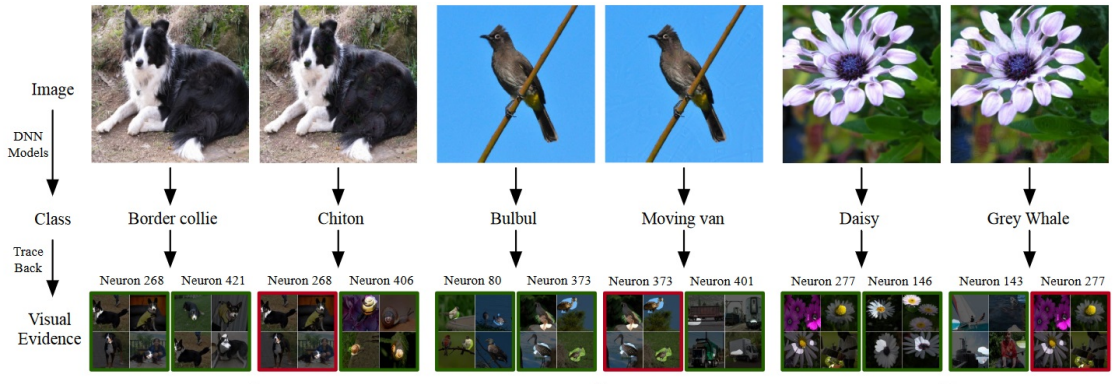


Figure 2.26: Demonstration of this prediction difference analysis. For each example, the top two most relevant neurons are displayed and the top four images that have highest activations. (Figure 10 of [DSZB17])

Though this method is not precise in terms of regions of the image as other forms of explanation generation are, this is still an interesting way of explaining how a neural network got to its decision.

2.4.4 Miscellaneous Techniques

Finally, we will look at miscellaneous ways of explaining how neural networks reach decisions. We will first consider architectures that give textual explanations of their decisions, usually involving recurrent neural networks [Bar17, DSZZ17]. Then we will look at neural networks from the perspective of network flow [DTZ17]. Though these explanation techniques are interesting by themselves, these are not visual explanations and so are not the main focus for this project.

Textual Explanations

Textual explanations for neural networks are typically a couple of sentences beginning with "This image is of a [class] because...". The first architecture that we looked at is InterpNET [Bar17]. InterpNET is partly inspired by how the visual cortex works in humans where it is used even when a person is asked to imagine. In the same way InterpNET uses the machinery of the network to guide its explanations.

The problem involves supervised classification and explanation where triples of input, classification and explanation are given. This is different from captioning which only involves an input and text where the text only explains the neural networks observation and not its reasoning. To generate the explanation in InterpNet, all of the internal activations of the network are concatenated and fed to a separate network for processing text. In this case it is LSTM (Long short-term memory) [HS97] network which has been successful when it comes to textual based tasks. An example model for the Caltech-UCSD Birds 200-2011 (CUB) dataset is shown in Figure 2.27.

Training works in a two stage method by first training the classifier part to convergence and

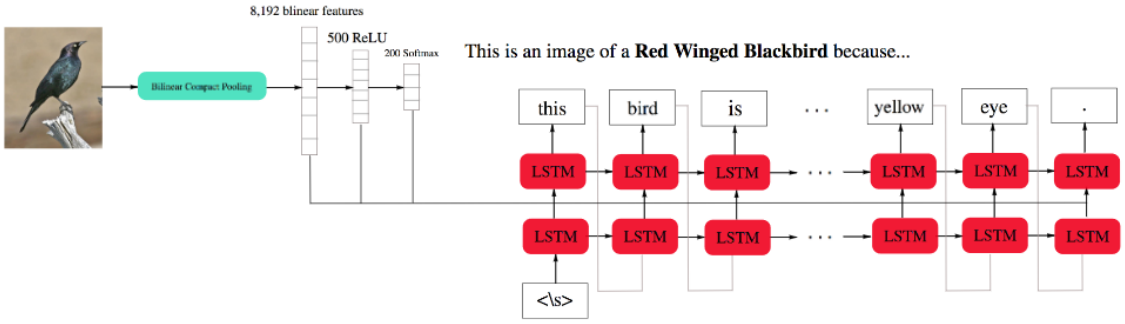


Figure 2.27: An example model using the Caltech-UCSD Birds 200-2011 (CUB) dataset. Notice that all of the activations of the network are then fed into the explainer module. (Figure 1 of [Bar17])

then the explainer to convergence [Bar17]. The explainer stops propagating the gradient at the feature vector input to avoid sacrificing accuracy of the classifier. Both are trained in the standard way of neural networks which is stochastic gradient descent with momentum. This architecture has high performance with higher scores in METEOR and CIDer (standard metrics) than the state-of-the-art baseline model. The InterpNet model with two hidden layers was also superior at the captioning task which provides evidence

As interesting as this architecture is, a whole recurrent neural network needs to be trained which would be very time consuming for the intended application. This, combined with the fact that the explanations are textual rather than visual, was why this method was not considered for further implementation.

The method in [DSZZ17] takes a different approach. It uses an ecoder-decoder stack instead. A stack of CNNs and bi-directional LSTMs are used to extract interpretable features from a video source. One could see this working with a single image by having a single CNN and LSTM pair. This is then followed by an LSTM as a decoder to the generate an explanation. To make the explanations more interpretable, latent topics are extracted from descriptions as semantic information and used to guide training with interpretive loss:

$$L_1(V, \boldsymbol{s}) = \|f(\frac{1}{n}\sum_{i=1}^{n} \boldsymbol{v}_i) - \boldsymbol{s}\|_2^2$$

where V is the collection of video vectors, f is an arbitrary function from features to topics and $\boldsymbol{s}$ is binary vector representation of topics. The topic model used is Latent Dirichlet Allocation (LDA) which has been applied to image, video and text analysis tasks before. This method's explanations in BLEU and METEOR scores were the best found in the methods it was compared to.

The author's illustrated a "human-in-the-loop" method of training [DSZZ17]. This works by having humans give the correct topic t when it givens an inaccurate output. The network then retrieves neurons associated with this topic using prediction difference [RŠK08]. The original features $\boldsymbol{v}$ are transformed into $\boldsymbol{v}*$ by adding the average of them in a subset of training videos containing the topic t. The network is then trained to reproduce $\boldsymbol{v}*$ instead of $\boldsymbol{v}$ so the neurons associated with the topic have higher activations.

This method was found to achieve better performance compared to various competitors in video captioning tasks in both METEOR and BLEU scores. Unfortunately, the process of training LSTMs and the CNN together would be impractical for this project.

Network Flow

Another way at understanding feedforward networks (like CNNs) is through analysing network flow [DTZ17]. It starts with the premise that there is no definite reason why particular classes

are misclassified more often than others in artifical neural networks. However, by analysing how information flows through the network itself, one can see that networks for particular classes that are very similar to each other are more likely to get misclassified then those whose flows are very different.

One can treat training of neural networks as developing these flows (called class-pathways) for each class. These are called class pathways as information is propagated through the network and finally flowing into the output (class) nodes in a standard feedforward neural network. Each class-pathway is represented by a collection of neurons in each layer of the network. An example of a class-pathway for a traditional feedforward network is shown in Figure 2.28. This works particularly well with ReLU units as neurons with negative neurons are completely cut out of the network so only about 25-30% of neurons are activated [DTZ17]. This means that there is a lower computational complexity and so is less likely to overfit.

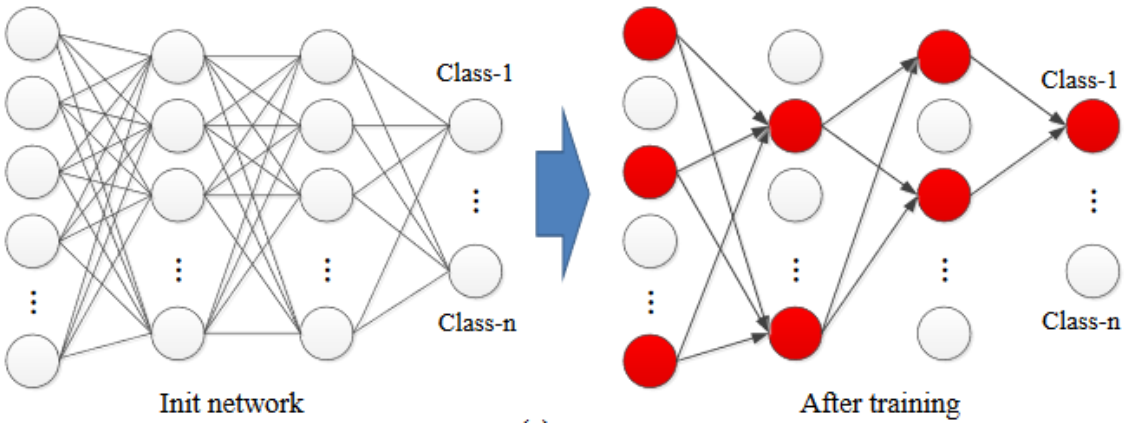


Figure 2.28: An example of a feedforward neural network and a class-pathway for class 1. Red nodes being those that are in the class-pathway. (Adapted from Figure 1 of [DTZ17])

The distance between two class-pathways for classes i and j is simply the euclidean distance of the activations:

$$\|D_{ij}\|_2 = \sqrt{\left(\sum_{index=0}^{N} (V_i[index] - V_j[index])^2 \right)}$$

Intuitively, the smaller distance between class-pathways, the higher the error rate of the two classes is likely to be. The class-pathways are calculated themselves by propagating the activations backwards from the output neuron connected to that particular class.

The authors of [DTZ17] observed that the distance between class-pathways do predict error rate. They trained a normal feedforward neural network on the MNIST data set and calculated each of the class-pathways. The classes corresponding to the digits 3, 8 and 9 had the lowest average class-pathway distance and 80.8% of all errors in the test dataset were predicted as one of these classes, supporting intuition. Also, for any class i, five of the closest class-pathways made up 91.61% of all incorrectly predicted examples. It was also found that one could decrease the complexity of neural networks by removing any node that had a small activation for all of the classes without increasing the error rate significantly.

Although this method looks specifically at model explanations by explaining each class, it could easily be extended to prediction explanations. This can be done with a forward pass on the input of the particular example and calculating the distance of the activations between each of the classes. However, since the only real visualisation is which nodes are activated which would overwhelm the user, this method will probably not be implemented in this particular project.

2.5 Synthesis Techniques

Now that we have discussed possible visual explanations, we will also cover image synthesis techniques. While some methods discussed look generate visual explanations from the network [Kin, SDBR14, ZF14], we will cover in this section specifically generating images for a given network.

We will begin by looking at methods that produce synthesised images based on a particular class, that typically lead to less natural representations. We will then consider visualisations that use natural images as priors in order to produce more natural looking representations.

2.5.1 Generating Images From A Class

We will begin by first looking at methods that synthesise images based off of a particular class, rather than generating based off a particular input image. The first method will look at generating an image that optimises an objective function [SVZ13] while the second looks at using the Hamiltonian Monte Carlo (HMC) method to generate visualisations for a class [DLW14].

Class Model Visualisation

For a given CNN (though this could be generalised to any classifier), one can generate an visualisation of a particular class c by finding an image numerically that maximises the score for that particular class [SVZ13]. More formally this is done by maximising an objective function and finding an L_2 regularised image such that the score is high:

$$\arg\max_I S_c(I) - \lambda \|I\|_2^2$$

where λ is the regularisation parameter. A locally optimal I can be found with the backpropagation method but changing the input image and keeping the weights fixed. The authors also added the training set mean image to the result [SVZ13]. The reason why the unnormalised score S_c was used rather than the output from softmax was because optimising the softmax output can be done by minimising it for the other classes.

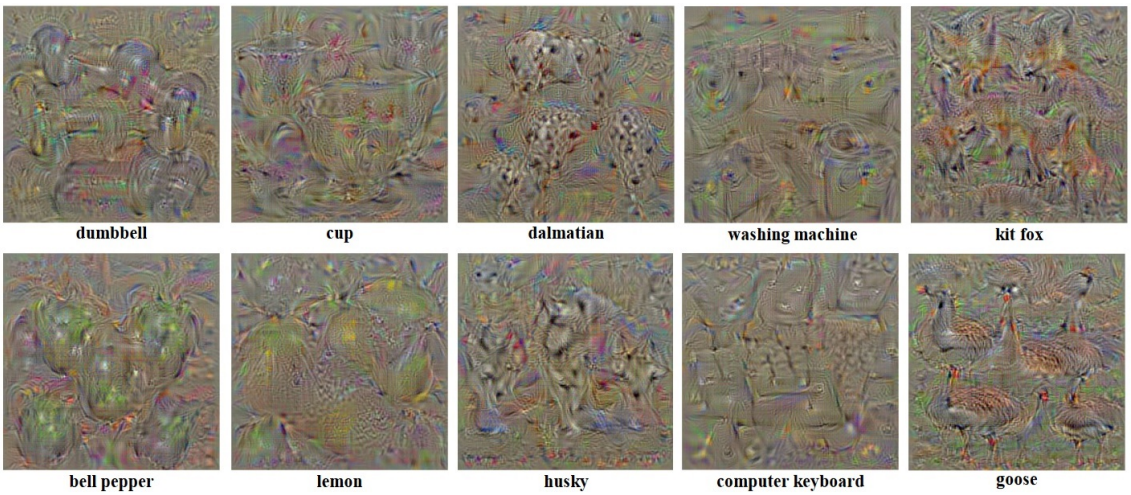


Figure 2.29: Examples of visualisations for various classes. Notice that multiple instances of these classes are produced from multiple angles. (Adapted from Figure 1 of [SVZ13])

A range of examples can be seen in Figure 2.29. For each class, different aspects of the class are captured in a single image. However, some classes such as the computer keyboard are very hard to make out. This is probably due to the fact that natural images are not used to influence the optimisation process, only L_2 regularisation is used.

Hamiltonian Monte Carlo

Another way to generate visualisations is to sample the images by using Hamilton Monte Carlo (HMC) [DLW14]. For this, one needs to consider the probability distribution on inputs x given category y and parameters learned w which would be $p_y(x; w)$. The parameters can be learned either discriminatively or generatively, the latter of which the paper goes into in detail.

For HMC, $p_y(x : w) \propto exp(-U(x))$ where:

$$U(x) = -f_y(x; w) + \frac{|x|^2}{2\sigma^2}$$

Where σ is the standard deviation of the reference distribution $q(x)$ and $f_y(x; w)$ is the score for the class y given image x and parameters w. In physics, x is the position vector while $U(x)$ is the potential energy function. To implement Hamiltonian physics, momentum vector ϕ is needed and the corresponding kinetic energy function being $K(\phi) = |\phi|^2/2m$ where m represents mass. Each iteration of HMC draws a random sample from the marginal Gaussian distribution of ϕ and Hamiltonian dynamics are applied that conserves total energy (kinetic and potential energy) [DLW14]. Like in the method in [SVZ13], the images produced for each class are noisy although they are more recognisable than the previous method mentioned. The final method discussed that generates images not only at the final layers but for any layer in the network and produces the most recognisable results we found.

Deep Visualisation

Another approach for synthesising more natural looking images to add even more regularisation terms to the objective function [YCN+15]. The method used for this is gradient ascent but with the regularisation applied as an operator. This means that the image $\boldsymbol{x}$ is moved in the direction of the gradient followed by the direction of the regularisation operator r_θ:

$$\boldsymbol{x} \leftarrow r_\theta\left(x + \eta\frac{\partial a_i}{\boldsymbol{x}}\right)$$

The paper includes four different regularisation terms

1. L_2 **decay**: The same as the regularisation term in [SVZ13]. This is used to prevent a small number of extreme pixel values dominating the image. These values neither occur frequently naturally nor are good for visualisation.

2. **Gaussian blur**: Gaussian blur helps to penalise high frequency information. This s information that causes high activations but is neither realistic nor interpretable. This blur is applied with θ_{b_width} as a hyperparameter. Since this function is relatively computationally expensive compared to the other regularisation methods, another parameter was added so it is applied at regular intervals instead.

3. **Clipping pixels with small norm**: After applying the first two steps, $\boldsymbol{x}^*$ has somewhat small and smooth values but also non-zero values everywhere. This means pixels not associated with the main object will also show some pattern as well which is not desirable. All pixels with a norm below a particular threshold θ_{n_pet} will be set to zero.

4. **Clipping pixels with small contribution**: Instead of clipping pixels with small norms, a smarter method is to clip pixels with a small contributions. Contributions can be found by checking the activation of a particular neuron before and after the particular pixel is set to zero. This is slow as a forward pass must be done for every pixel so a different method is instead to estimate by an element-wise multiplication of the image and its gradient.

Combining these regularisers leads to more natural images than the previous methods discussed. Examples of these visualisations are shown in Figure 2.30 when looking at the final layer. In particular, colours are much more visible, especially compared to the visualisations in [DLW14]. It was thought that neural networks would not produce such natural looking images because neural networks use discriminative features such as the pattern of spots on a leopard to classify an image.

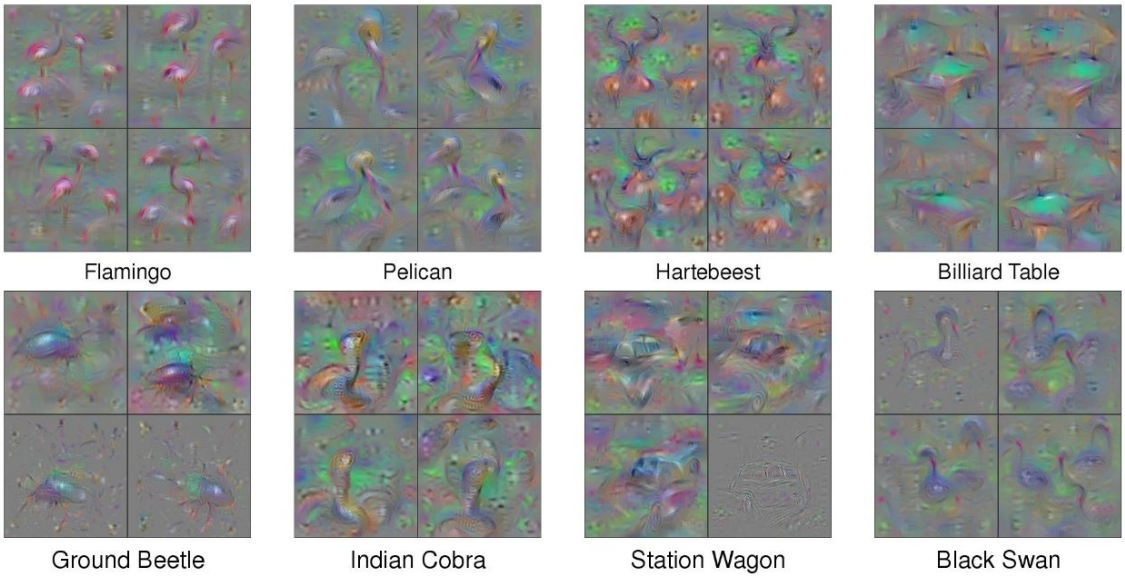


Figure 2.30: Examples of visualisations for various classes using Deep Visualisation. Colours are very clear compared to previous methods such as the pink of the flamingo. (Adapted from Figure 4 of [YCN+15].)

However, the authors propose that the reason more successful results were not produced previously is because the priors were just not strong enough [YCN+15].

To conclude this background section, we will look at synthesising more natural images for a particular prediction, especially with natural pre-images.

2.5.2 Synthesising Images From Prediction

Since we looked at generating images based on the particular class, we will at synthesising images based on the encoding of a particular image. Although these cover reconstructing the whole image and the project looks at specific regions and features, the methods here could be adapted for this usage. One way would be to find the neurons that encode the particular feature and then maximise those activations rather than the whole encoding. We will look at a general method for reconstructing images [MV15] as well as different reconstructions such as caricaturisation [MV16].

Reconstructing Images from their Encodings

When an image is run through a trained CNN in a forward pass, information from the image is encoded into the network with the activations of neurons. What the authors in [MV15] attempt to do is approximate an inverse function of the given internal representation of input $\boldsymbol{x_0}$ which is Φ_0:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \ell(\Phi(\boldsymbol{x}), \Phi_0) + \lambda \mathcal{R}(\boldsymbol{x})$$

where ℓ is the loss function between two representations. In this case, the euclidean distance was used but is normalised by Φ_0. Here, $\mathcal{R}$ is a regulariser capturing a natural image prior. Minimising this equation leads to $\boldsymbol{x}^*$ that resembles the original image from the viewpoint of the representation. Examples of these reconstructions are shown in Figure 2.31. The first regulariser used is the alpha-norm $\mathcal{R}_\alpha(\boldsymbol{x}) = \|\boldsymbol{x}\|_\alpha^\alpha$. By choosing a relatively large exponent, such as $\alpha = 6$ which is what the authors used, the range of the image is restricted so it does not diverge [MV15]. Another example is to use the total variation (TV) as a regulariser:

$$\mathcal{R}_{V^\beta}(\boldsymbol{x}) = \sum_{i,j} \left((x_{i,j+1} - x_{ij})^2 + (x_{i+1,j} - x_{ij})^2 \right)^{\frac{\beta}{2}}$$

Using TV as a regulariser with $\beta = 2$ removes "spikes" from the reconstruction as well. Gradient descent with momentum is used to optimise the objective function.

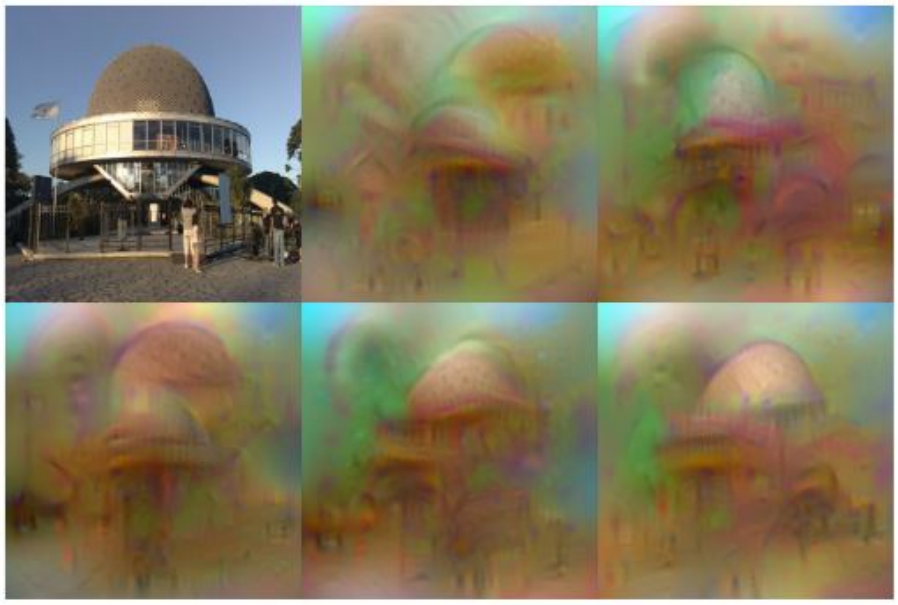


Figure 2.31: Examples of various reconstructions of the image in the top-left corner. To the network, these are all roughly equivalent. (Figure 1 of [MV15].)

At deeper layers in the network, more and more variation was observed. This would be useful for the feature synthesis tool as one can generate different representations of a particular feature e.g. a cat's ears easily.

Activation Maximisation and Caricaturisation

The authors of [MV15] also built on this work and considered two other types of visualisations [MV16]:

1. **Activation maximisation**: Here, $\boldsymbol{x}^*$ is found that maximally excites a particular component of the network such as a neuron. This helps a person to understand a particular component's function or meaning.

2. **Caricaturisation**: Initial image x_0 is modified so that the representation $\Phi(\boldsymbol{x_0})$ is maximally excited. This emphasises the meaning of combinations of representation components that are active together.

An interesting method that this paper covers in particular is only reconstructing to a particular subset of the representation by using a binary mask M in the loss function:

$$\ell(\Phi(\boldsymbol{x}), \Phi_0; M) = \frac{\|(\Phi(x) - \Phi_0) \odot M\|^2}{\|\Phi_0 \odot M\|^2}$$

This could be used for the feature synthesis tool of the project by finding the components associated with the feature and then masking the rest out.

As well as the alpha-norm and TV regularisers discussed in [MV15], [MV16] also introduces a jitter regulariser that randomly shifts the input image before generating its representation. This jittering counterbalances the downsampling in the earlier layers of deep CNNs and so the resulting images are crisper.

For optimisation, $\boldsymbol{x}$ is initialised as i.i.d. noise so that the only information considered is the representation of the original image Φ_0. This means that the differences of the reconstructions obtain insights into the invariances of the representations. Reconstructions are evaluated quantitatively by first checking the distance between their representations and the original image representation.

When it comes to caricaturisation, discriminative features (such as the fox's head) are repeated several times as can be seen in Figure 2.32.

These two papers [MV15, MV16] seem to give a good basis for the feature reconstruction tool of the application.

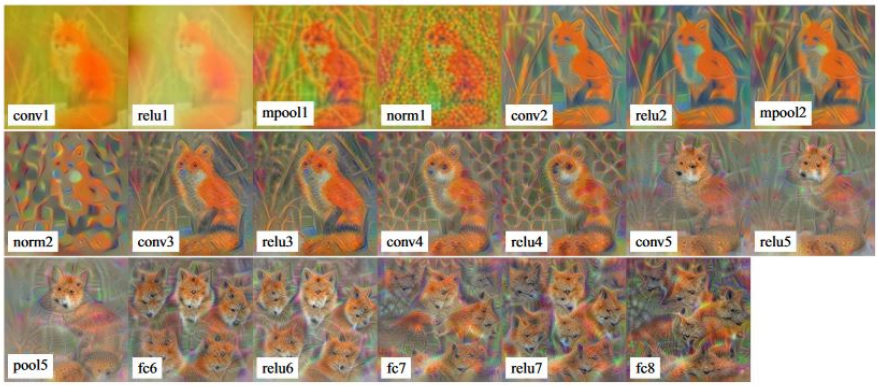


Figure 2.32: Caricatures of a "red fox" image at different layers of the network. Notice that at deeper layers multiple foxes start appearing. (Figure 20 of [MV16].)

Chapter 3

Feature Occlusion

Now that we have covered the main web application and how the user would upload the models that they wish to test as well as their prediction images, we will now cover the two explanation methods devised and implemented for this project. The first of which is what we have called feature occlusion.

3.1 What is Feature Occlusion?

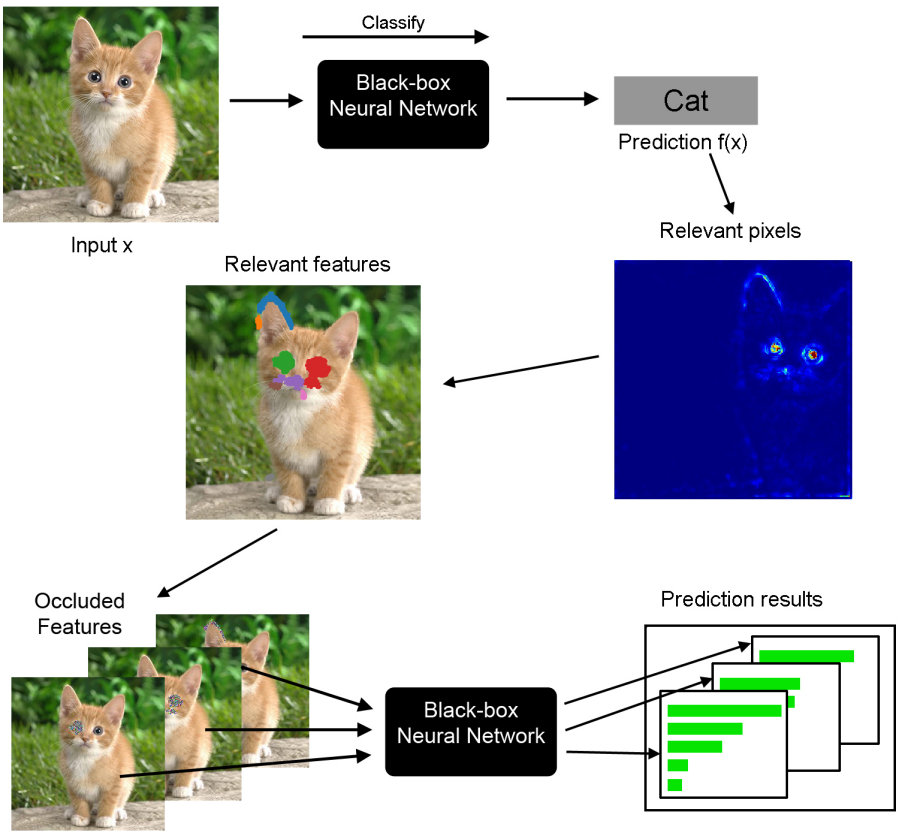


Figure 3.1: The entire pipeline for one example of the feature occlusion tool. First of all, the particular image is passed through the network to get a particular prediction, in this case "cat". Then the relevant pixels are obtained using Layer-wise Relevance Propagation [BBM+15] and these are clustered into features. Finally, multiple images with features occluded are fed back into the network and prediction results are compared.

Feature occlusion, simply, involves finding relevant features for a given neural network model and input (in this case an image), then looking at how removing these features changes the predic-

tions that the network has for the resulting image. This helps to explain neural networks in three ways:

1. **Shows which features of the input are most relevant** When trying to explain neural network classifications, it is important to first find which features are relevant. For feature occlusion, this is done by the initial stage of feature extraction. By occluding each feature individually, we can see which single feature makes the largest change in classification from the original class and see which features are more important. We can also do this the other way by occluding all of the features found and then seeing which features would lead to the original class being predicted. These different perspectives are discussed further in the Metrics section.

2. **Shows features of an input that may not be relevant to the model for a particular class** The feature extraction not only tells which features are the most relevant but also those that are not relevant. There may be large areas of the input that are completely ignored by the network for a particular class. For those features that have been selected, we can also see if they are not that relevant by observing a small change in classification when occluding them.

3. **Explains how combinations of features interact with each other** As well as looking at the changes to classifications that features have in isolation, it also possible to look at how classifications change when multiple features are occluded. An example for an image of a dog may be its eyes and ears. A model would probably be robust to predicting a particular breed of dog if just one of its eyes and one of its ears are occluded. However, if both eyes are occluded then the model would completely lose a part of the information needed to classify an image. If we start off with all of the features occluded, then a single feature being revealed may not be sufficient to predict the correct though revealing multiple features might. If only the eyes of the dog are visible, the network could mistake the image for a wolf, but revealing the (usually) floppy ears of the dog would probably make it less confident in predicting a wolf.

We will next discuss why we chose feature occlusion as an explanation method for convolutional neural networks. Next, we will guide the reader through our method for extracting relevant features using a combination layer-wise relevance propagation [BBM+15] and a modified flood-fill algorithm as well as different strategies for occluding these features. Finally, we will then cover interesting summary metrics that can be calculated automatically without the user's input.

3.2 Why Use Feature Occlusion?

Now that we have covered the basics of what feature occlusion, we will briefly outline why we decided to choose this as an explanation tool. We found this to be a great explanation tool for five main reasons:

1. **Interpretable** There are some explanation methods that dive deep into individual neurons in the network [DSZB17] or look at explanations between classes [DTZ17]. Though these may reflect the model very well, they are not the most interpretable, meaning that they can be difficult for humans to understand clearly. Since feature occlusion shows which features are important directly on the input, in this case the image, it is very easy for humans to understand these explanations. For example, if occluding a features produces a large positive change in favour of a particular class, then the user can intuitively reason that the particular feature was negative information for the particular class.

2. **Faithful to the model** If an explanation technique is not faithful to the model, what is the point in using it in the first place? As well as being interpretable for humans, feature occlusion also does not sacrifice much faithfulness to the model. There is some bias that is introduced when occluding features, however, but this is unfortunately inevitable as it is not possible to mark neurons in a network as "unknown".

3. **Does not require additional architecture** A problem with a few of the explanation techniques that we researched is that they would either require additions to the network

architecture [Bar17, SDBR14] or they might require training new components [Kin, DG17]. This is not only very time consuming (one has to train a new network) but may also modify the original network to the point that the technique is not faithful Feature occlusion fortunately does not have this problem as features are extracted directly from network weights and activations and occlusion only modifies the input to the network.

4. **Fast** Once the relevant features are found, which itself can be a fairly fast process, the explanation technique is very quick. This is because the only two stages are to occlude the features of interest and then feed the new input into the network. This great for the user because they are able to analyse a large number of inputs and models fairly quickly and get responses back in near real-time.

5. **Builds on existing explanation techniques** Feature occlusion uses both an implementation of Layer-wise Relevance Propagation (LRP) [BBM+15] as well as principles from Prediction Difference Analysis [RŠK08, ZCW16, ZCAW17]. These techniques are well known have been expanded upon since. This gives us confidence that feature occlusion will also be a useful explanation technique as it is built upon existing successful techniques.

3.3 Extracting Relevant Features

For feature occlusion, the first stage is to extract from the network the features that are relevant. However, we have to first determine which features would be useful and relevant. One basic idea could be to take a patch of a fixed size (typically a square) and then look at all of the possible patches in the image of that size. This would be similar to prediction difference analysis covered in section 2.4.2 [ZCW16, ZCAW17] or the occlusion testing in [ZF14]. There are two main problems that we found with this approach:

1. **A lot of features have to be covered** Since we would be covering every patch. For a 224x224 pixel image (input size for the VGG networks [SZ15]) and a patch size of 9x9, 216x216 $>$ 46000 patches would be considered. This is not only computationally expensive (a forward pass is required for every single one of these patches) but it also difficult for users to comprehend such a large amount. This can be alleviated somewhat by summing the prediction differences in a 2D matrix [ZCW16, ZCAW17] as seen in Figure 2.25. However, this is still too time consuming for our intended goal of using this in a web application.

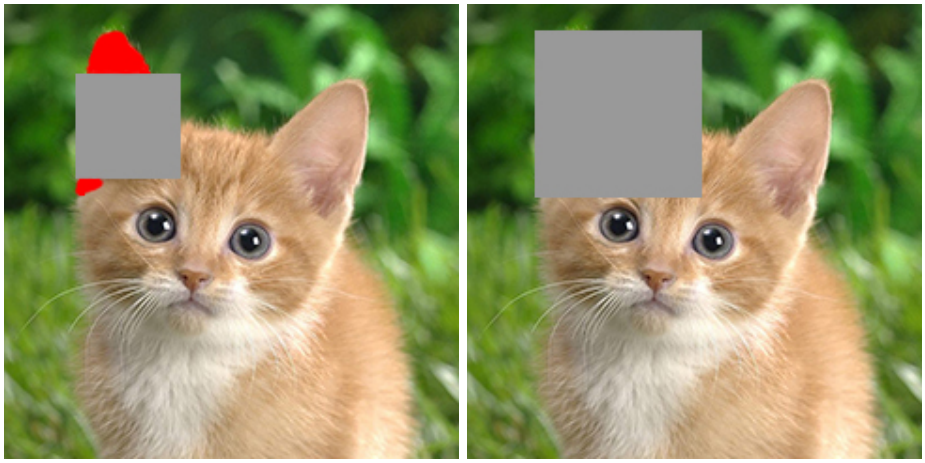
2. **Not the entirety of a particular feature will be occluded** Another issue that is illustrated in Figure 3.2 is that these patches can end up not occluding the entire feature. A square patch, for example, would not be able to cover an oblong feature such as an ear. This could be remedied by using larger patches, but then multiple features at once may be occluded which is also not satisfactory. Combining multiple patches would also not work, not just because of the exponential number of combinations that need to be tried but because of a lack of context which we will cover briefly now.

If we were to consider every patch of a fixed size, it would be fairly easy to present a human with a few of these patches so they could find a particular feature such as an eye or ear. This is only possible because a human has the appropriate context i.e. they already know what an eye or ear is supposed to look like. Neural network also have this too as it has been shown that individual neurons can learn different high level features [Le13]. This means that we end up in a chicken-and-egg scenario as we have to be able to determine which features a neural network knows in order for the network to tell us which features of an image that it thinks are relevant.

We found that a much better method for extracting the features is finding the pixels of the image that the neural network found the most important, then grouping these pixels into higher level features.

3.3.1 Layer-wise Relevance Propagation

The first stage of extracting features is too find the most relevant areas of the input. Since the input is an image in this case, then it is looking at the most relevant pixels. As discussed in the background section 2.4.2, there are many different methods with which to achieve this. The method chosen should satisfy three properties:



(a) (b)

Figure 3.2: Example showing a problem with using square patches. The square patch in (a) is too small to cover the entirety of the ear (in red). The square patch in (b), however, may cover the entire ear but also covers a large amount of the top of the head.

Table 3.1: Comparing the properties of various methods that try to determine the relevance of pixels

Method	Fidelity	Computation Required	Discrimination in Features
Sensitivity Analysis[SWM17]	Low	Low	Medium
Guided BackPropagation[SDBR14]	Medium	Medium	Medium
DeConvNet[ZF14]	Low	Low	Medium
CNN Fixations[MGB17]	Medium	Medium	Low
Layer-Wise Relevance Propagation[BBM+15]	Medium	Medium	High
PatternAttribution[Kin]	High	Very High	High

1. **Fidelity to the Model** The most important property for methods that determine relevant features (and any other explanation method) is how well the method reflects the model being analysed. It is no use to have a method that extracts features, no matter how useful they are, if they are not features that the model itself thinks are relevant. Otherwise, it would not be a useful analysis as one cannot determine the strengths and weakness of the model. Some of the methods discussed in the background have very high fidelity (such as PatternAttribution [Kin]) while others have low fidelity (Sensitivity Analysis [SWM17]).

2. **Computationally Inexpensive** Since the resulting tool needs to be implemented in a web application, the method to calculate the relevances must not take too long to process. Since there will be multiple users with multiple models and images to analyse, the method should also not use too many computational resources (memory, CPU).

3. **Discriminative in Features** Once we have the relevances for each pixel, we then want to be able to then construct the higher level features such as shapes, facial features and parts from these pixels. The easiest way to do this is if these features are separate from each other. In this case, regions of relevant pixels are separated by less or not relevant pixels. This property is not strictly required but it allows us to separate the individual features easily.

We found Layer-wise Relevance Propagation (LRP) [BBM+15] to be the best compromise of these three properties. It is the second-most faithful method to the model, with PatternAttribution [Kin] being the most as discussed in the background. It is much more faithful than a method like Sensitivity Analysis[SWM17] which does not really look at relevances at all but the derivatives of each pixel i.e. which pixels will produce the largest change for the particular class. However, LRP has an advantage over PatternAttribution in that it is much less computationally expensive. Once the signal estimator is trained, PatternAttribution is quite computationally inexpensive but training can take several hours even when using 3-4 GPUs. For the 16 layer VGG network [SZ15],

we found that it only took around a minute to calculate relevances using LRP on a single CPU. This is because LRP requires only a single forward and backward pass. Since the results are deterministic for a single model and image, the results can be saved as well. It is also a much more discriminative of features than for example CNN fixations[MGB17], for example. This is mainly because CNN fixations method only finds a few relevant points and a heatmap is generated to estimate the other relevant pixels.

How LRP works

As discussed in the background section, LRP works on the basis of preserving the relevance across

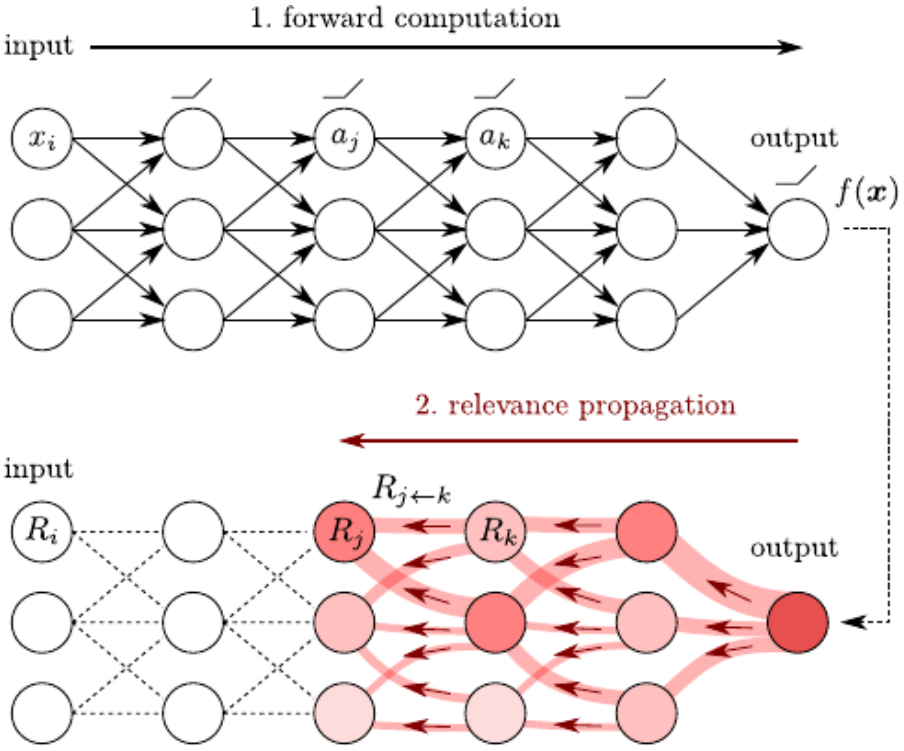


Figure 3.3: A diagram showing how LRP works with the forward pass and first three steps in the backward pass. The red arrows indicate the relevance flowing with more opaque reds and thicker lines indicating larger relevance. (Figure 11 of [MSM18]).

layers of the network. To start with, a forward pass is completed for the particular input example to obtain the activations of all of the neurons. Then, the relevance for the neuron corresponding to the particular class is set to a constant value (usually 1). This relevance is then propagated backwards through the layers until the first layer is reached where the relevances of one layer are calculated in terms of the relevances of the next layer. This process is demonstrated in Figure 3.3 The values of the relevances at these neurons correspond to the relevances of the individual pixels of the input image. The relevances being preserved means that there is no external relevance that is added to the computations, ensuring that any relevance comes from the model itself. For a particular layer the relevances R_j are calculated as follows: The standard redistribution rule of relevance from one layer $l+1$ to a neuron in the previous layer l is as follows:

$$R_j = \sum_k \frac{x_j w_{jk}}{\sum_j x_j w_{jk} + \epsilon} R_k$$

where x_j are activations of layer l, R_k are relevance scores for neurons at layer $l+1$ and $w_j k$ is the weight from neuron j to neuron k. The alternative rule that has been implemented separates

positive and negative activation and weight pairs and have difference coefficients α and β:

$$R_j = \sum_k \left(\alpha \cdot \frac{(x_j w_{jk})^+}{\sum_j (x_j w_{jk})^+} - \beta \cdot \frac{(x_j w_{jk})^-}{\sum_j (x_j w_{jk})^-} \right) R_k$$

where $\alpha - \beta = 1$. Here we used $\alpha = 2$ and $\beta = 1$. These equations are applicable both for the fully-connected layers and convolution layers and can be done efficiently using matrix-vector and element-wise multiplications. The first rule (also known as the z^+ rule) can be done in the following steps: [MSM18]

$$\begin{aligned} \mathbf{z} &\leftarrow W_+^T \cdot \mathbf{a} \\ \mathbf{s} &\leftarrow \mathbf{R} \oslash \mathbf{z} \\ \mathbf{c} &\leftarrow W_+ \cdot \mathbf{s} \\ \mathbf{R} &\leftarrow \mathbf{a} \odot \mathbf{c} \end{aligned}$$

where $\odot$ and $\oslash$ are element-wise multiplication and division respectively and $\mathbf{a}$ is the set of activations of the particular layer. This can be easily extended to the $\alpha\beta$ rule by just separating by weight values. Both the first and third computations can be completed through forward and backward computations of the layer respectively.

For pooling layers (common layers in convolutional neural networks), relevances are distributed proportionally given by the following rule [20117]:

$$R_i = \sum_j \frac{x_i \delta_{ij}}{\sum_i x_i \delta_{ij}} R_j$$

Where δ_{ij} indicates whether neuron x_i is in the pool x_j. These computations can also be done using forward and backward pooling operations.

3.3.2 Clustering Algorithms

We now have a way to determine the relevant pixels for an image to a reasonable degree. Unfortunately, there are a large number of pixels and there is no use having the user occlude them individually. Therefore we need to cluster these pixels into higher-level features. The pixels in these clusters should both be in fairly close proximity with each other as well as reflect higher-level features when clustered. The latter is more qualitative so human evaluation is required to determine whether the algorithm is working correctly. We tried two different clustering algorithms. The first being a k-means clustering and the second being a modified version of the flood fill algorithm which is most well known for being used as the "paint bucket" tool in graphics programs. We found that the flood fill algorithm was much more useful because of the dynamic number of clusters.

K-means clustering

One way to cluster items is by choosing an arbitrary number of clusters and then fitting the items so that they belong to the cluster whose mean (centroid) they are closest to. This is known as K-means clustering for a fixed number of clusters k. In this case we used the Clustering package from SciPy[JOP+] and the `kmeans` function after whitening the coordinates of the pixels so that they are normalised. The centroids are adjusted in order to reduce the distances between the pixels and their cluster centroids. As can be seen from the examples in Figure , the variability of clustering is very high. In short, k-means clustering does not work at all for a fixed k. This is due to the variability of the number of features that would be found for a particular. Compared to the image of the fox, there are a lot more distinct features that a human would identify as relevant for the cat given the output from the LRP algorithm. A possible solution is to instead try clustering on multiple values of k and selecting the clustering makes the most sense. There are two main problems that we found in this approach:

1. **Hard to determine what is a good clustering result** This is a similar problem to the chicken-and-egg situation discussed earlier. In order to determine a good cluster result without human input, we already need to know what are the relevant features which we are trying to obtain in the first place.

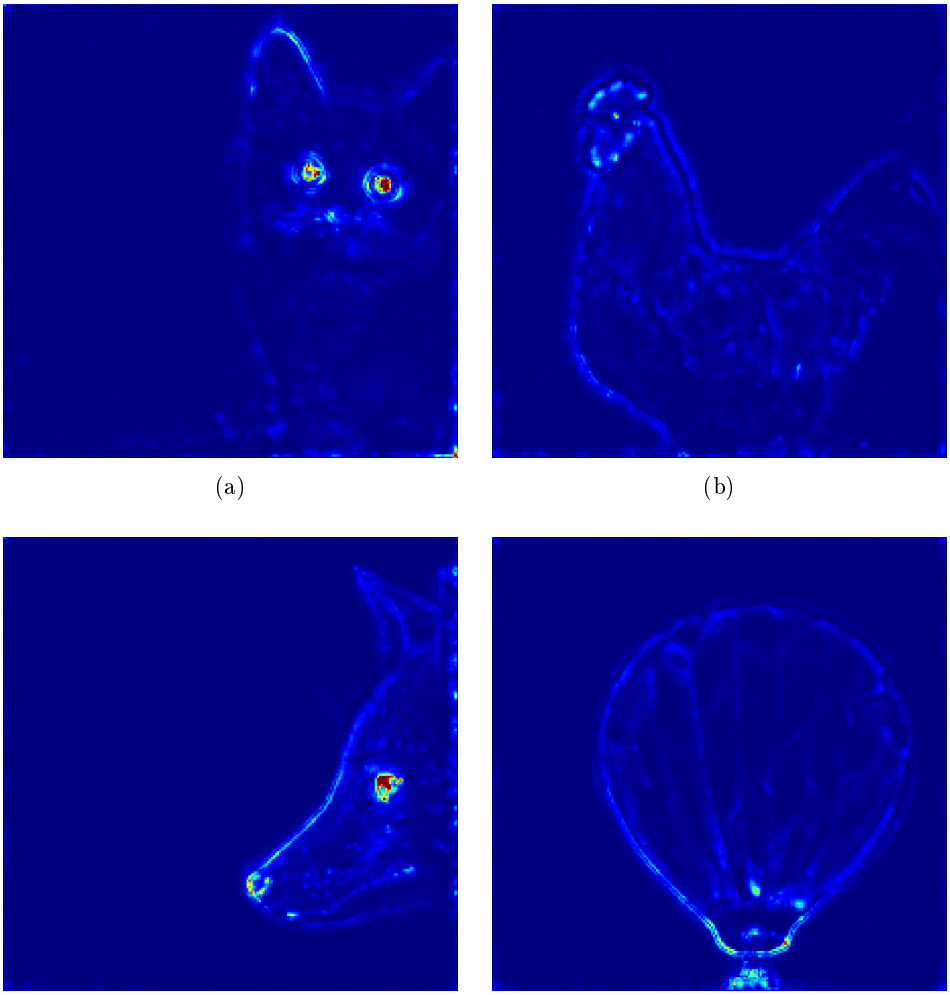


Figure 3.4: Four results from LRP calculations. In order, (a) a cat, (b) a rooster, (c) a fox and (d) a hot air balloon. Notice that for the animals the eyes are very important and for the hot air balloon its basket.

2. **Two separate features can be physically close to each other** K-means clustering works by reducing the distance between pixels and the clusters' centroids. This can result in very strange results in certain cases. An obvious example would be for images that have features close together. A person or animal's noses and eyes are typically close to each other, especially in a photograph taken face on. If, for example, an animal's nose and eyes are seen as relevant features. If these are the only relevant regions, then it's very easy to cluster the features when $k = 3$. However, if a feature more far away (such as its tail) is also relevant then clustering for $k = 3$ is not sufficient as two of the facial features (maybe all three) have to be merged together. The opposite problem occurs where k is too high and so features are split into multiple clusters.

So in order to obtain good clustering, we have to already know the number of features that the network has found as relevant. Due to these problems we could never get particularly satisfying results so we decided to try a different approach which would determine the number of clusters automatically by the nature of the algorithm.

Flood fill

This algorithm being a modified version of the flood fill algorithm. The flood fill algorithm would be familiar to those who have used "paint bucket" tools in graphics programs or played games like Minesweeper. For a certain point, the algorithm determines which points are connected to it

(a) (b)

Figure 3.5: Two cases where k-means clustering can go wrong. The first being where $k = 3$ and the rooster's eye, beard and comb are the same cluster. The second showing when $k = 6$, distinct features (such as the rooster's comb) are split into multiple clusters.

by looking at the points adjacent to it (by whatever definition of adjacent is suitable) and then to points adjacent to those and so on. For paint bucket tools, these pixels are restricted to those that are of the same colour. In our case, we modified the algorithm so any pixels with non-zero relevance are considered valid and therefore we would find separate "islands" of relevant features.

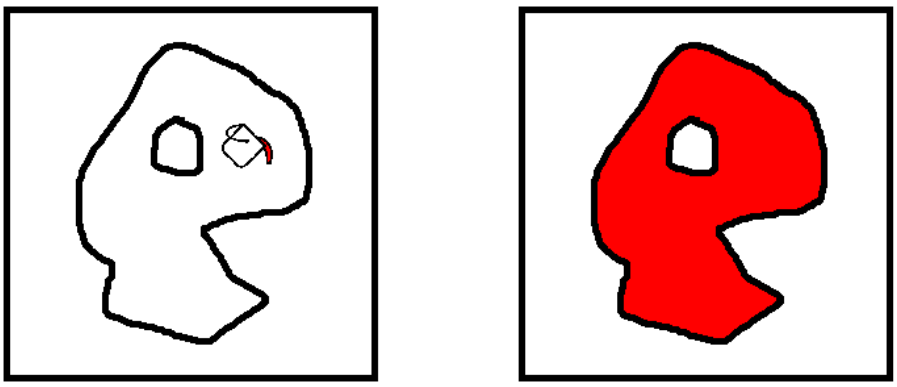


Figure 3.6: A diagram showing how flood fill works in a graphics program. User selects a pixel to colour red and all adjacent pixels of the same colour (white) are turned to red.

Initially what we found when running the algorithm were both very large features that would be made up of of a number of smaller high level features and very small features that did not be very relevant. Therefore we modified the algorithm further. The first thing we did was add a minimum threshold for pixels to be relevant. After trying various values we found that a number around $\frac{5}{HW}$ where H and W are the height and width of the image respectively was the most appropriate. It had the best balance between losing too much information and having large "blobs" of features. The next problem we had was we would have many very small features even though these were very close to each other. We found that there was typically a couple of pixels "gap" between these features so we also modified the algorithm so that instead of just considering pixels directly adjacent as part of the same cluster, we would consider all those within a small distance. Finally, we would still occasionally have very small clusters of a few pixels that were separate from the other features. We decided to instead ignore these clusters as they would not produce a substantial change in classification if they were occluded so we introduced a minimum number of pixels per cluster and we found 20 to be a suitable number.

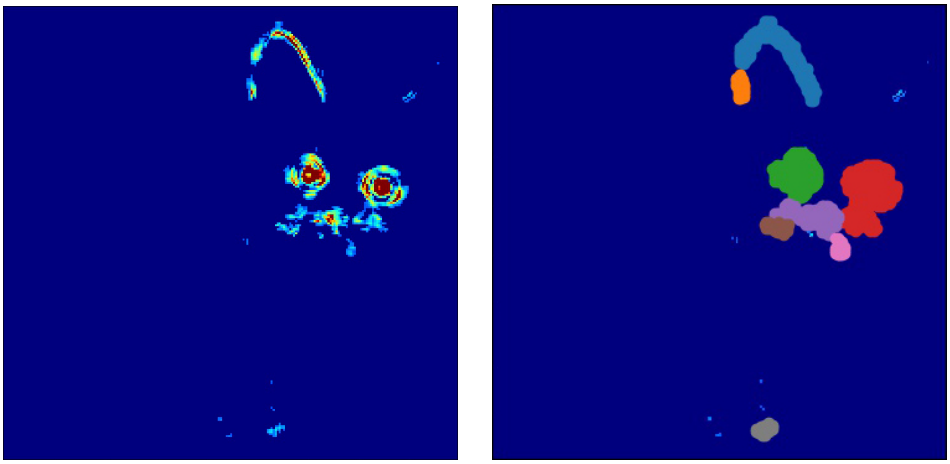


Figure 3.7: A demonstration of how flood fill works for obtaining features. On the left is an example of relevances from LRP. On the right are the eight individual "islands" of relevances that are obtained by the flood fill algorithm.

Overall we found that this modified flood fill algorithm was much better at clustering than k-means as we did not have to worry about the number of clusters and it faster to execute as there is no need to determine where the means of the clusters should be.

Figure 3.8: Examples showing the effect on features found according to the threshold. Each example shows a different threshold where threshold $= \frac{x}{HW}$ where $x = 1, 2.5, 5$ and 7.5 respectively. Maximum distance is 1.5 pixels and minimum cluster size is 20.

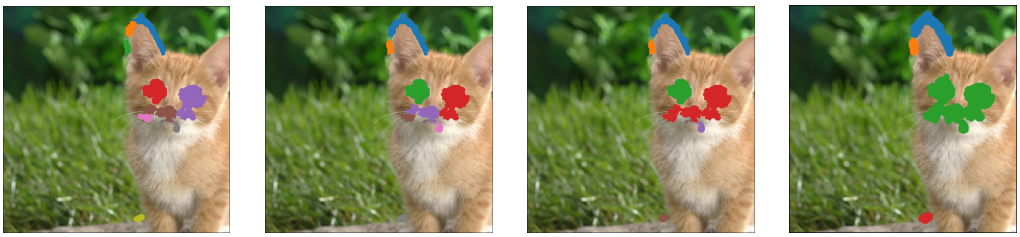


Figure 3.9: Examples showing the effect on features found according to the distance. Each example shows a different maximum distance which is $1, 1.5, 2$ and 3 respectively. The threshold is $\frac{5}{HW}$ and minimum cluster size is 20.

3.4 Occluding Features

Now that we have obtained the features that the network finds relevant, the final part is deciding how to remove them from consideration of the network when it predicts labels. Unfortunately with neural networks, it is not possible to just "ignore" neurons as each neuron depends on its incoming connections, the only difference being that the value is set to 0 instead of the original value. That means we have to find some way to change the pixel values to occlude the features. We have tried

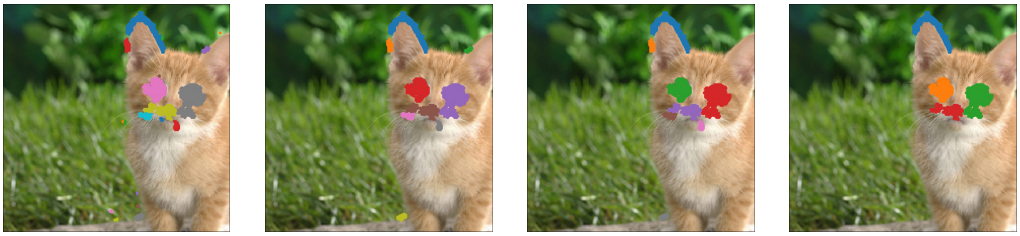


Figure 3.10: Examples showing the effect on features found according to the minimum cluster size. Each example shows a different minimum cluster size which is $0, 10, 20$ and 50 respectively. The threshold is $\frac{5}{HW}$ and the maximum distance is 1.5 pixels.

a few different methods and found that the ideal method of occlusion should possess these three properties:

1. **Little to no colour bias** An obvious problem with occluding pixels by changing their value is that it may introduce a new bias to the input. An obvious example would be colour. If the pixel values are replaced with a constant colour such as green, this would bias the network more towards objects with green in them.

2. **Little to no shape bias** Even removing bias in colour does not completely remove all biases. Even if the pixels in question are replaced with completely random noise, the shape of that feature would still be there as the noise contrasts with the rest of the image. This is particularly a problem if it is part of the contours of the object that are being occluded.

3. **Quick to calculate** Because the user will be experimenting with which features to occlude a lot, the process should be as computationally inexpensive as possible. This especially true due to the fact that the resulting image then has to be run through the network itself which can take up to a few seconds.

3.4.1 Individual Pixels

In order to avoid introducing bias towards a particular colour when occluding features, we decided to give each of the individual pixels a uniformly sampled random colour. This is not computationally expensive at all with modern libraries for sampling random values, though increased entropy does require more computation time. A slight problem with this occlusion method is that it is not deterministic as pixels are randomly sampled each time. However, we found that there was not a significant change in prediction values and the order of the top-5 labels were preserved. We did find though that a couple of examples, even when all of the features were occluded, would have little change in the prediction values. It could have been that the network was very robust to these examples but we decided to look into it further in case biases were introduced.

Figure 3.11: Occlusion by giving a random value to each pixel to be occluded. **Left**: The original image and prediction values. **Centre**: Left eye occluded and the changed prediction values. **Right**: Second case of the left eye being occluded with different values. Notice that although the sets of prediction values for the last two are not identical, they are not significantly different either.

Shape bias

An interesting property quickly found with this method after trying out a few examples is shape bias. An example where this is very obvious is for the image of a starfish as shown in Figure . Despite the fact that each of the starfish's arms have been occluded to some degree, the network still predicts that it is an image of a starfish with a 97% accuracy. It could just be that this network is very robust to starfishes which is a possibility due to their distinct shape. However, another explanation could be the fact that the outline of the starfish is still preserved due to the contrast between the noisy occluded parts and the sand background. We decided to investigate this further by looking at another occlusion method to break up the shape: minimum bounding rectangles and pixelisation.

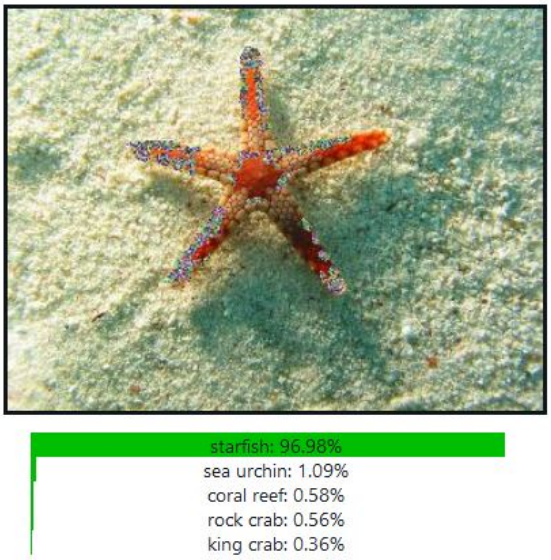


Figure 3.12: An example where it is mostly the outline of the object occluded. In this case it is a starfish and it still has a prediction of $\tilde{9}7\%$ even though every arm is occluded to some degree. This hints that there may be a shape bias introduced by this occlusion method.

3.4.2 Bounding Rectangles

3.4.3 Pixelisation

The other occlusion method that we looked at was pixelisation. This is similar to the method used by media to censor parts of an image, usually people's faces. This is done by taking patches larger than a single pixel and taking an average of the pixel values. We modified the algorithm slightly by also replacing the feature pixels with random noise like in section 3.4.1 in order to remove the colour bias. Our algorithm then goes over each square patch of a particular size and then average the pixel values for any patch that contains at least one of the pixel values. This produced better results for the starfish image where the prediction confidence for "starfish" dropped to 60% when all of the features are occluded.

Optimisation

Though this method works quite well, removing some of the shape bias that occurred with the original method, it is very slow. In fact, when looking at all of the features in the starfish example, it took a whole minute to occlude everything. Considering that the prediction itself is eight seconds, this method is very inefficient. Therefore, we went through a couple of optimisations. The first was to change the step of the algorithm that checks if any of the feature pixels are in the patch. This checks if any of the pixels in the feature . This is very time consuming as there are $n \cdot k^2$ comparisons where n is the number of feature pixels and k is the size of each patch for each patch. This was reduced significantly by instead checking if every feature pixel is within the range

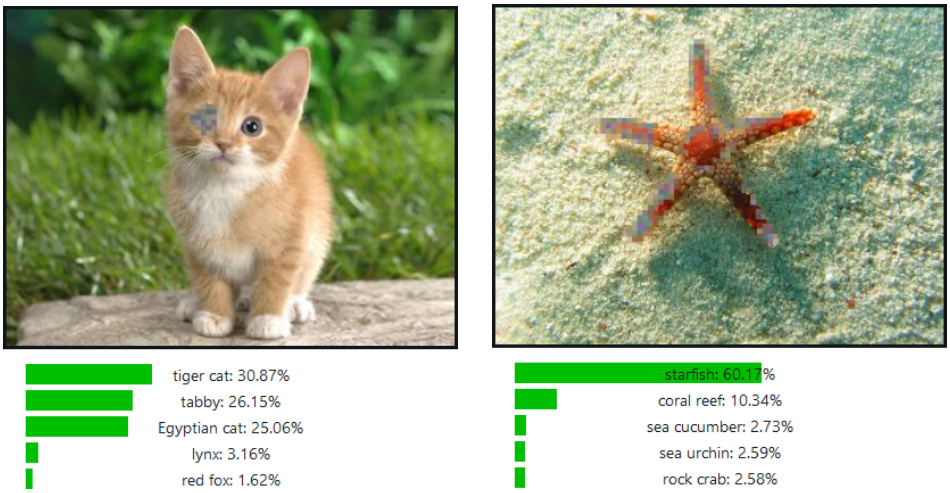
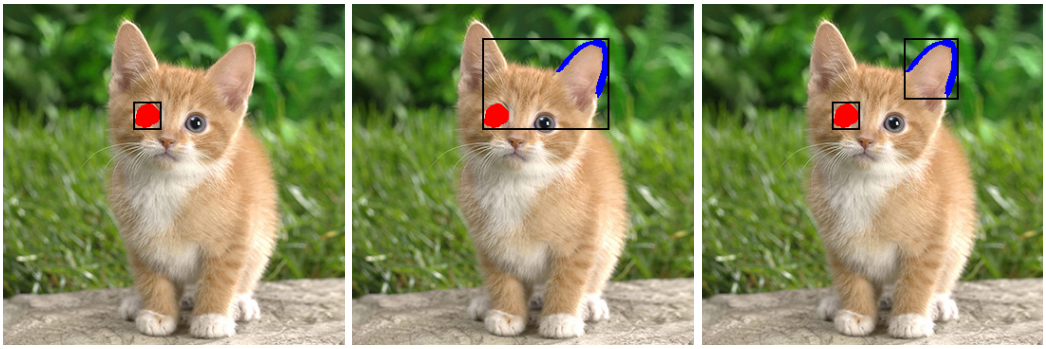


Figure 3.13: Two examples of the pixelisation method of occlusion. **Left**: The cat image with the left eye occluded and is similar to that in Figure 3.11. There is not much change in the prediction values compared to those in that figure, which is expected as we are only trying to remove shape bias. **Right**: The starfish image with all of the features occluded. In this case there is a drastic change in the prediction value compared to Figure 3.12.

of the patch. This reduced the time twenty-fold to three seconds. However, this is still nearly half the amount of time for the prediction.

We then decided to look at a smaller range of patches. Obviously we would only need to concern ourselves with patches that might include the features themselves, so we only considered those patches that are within the bounding box of the pixel features. An example is illustrated in Figure . This reduced the time further to around 0.5 seconds but that would depend on the features selected. Features that are far apart would result in a larger bounding box so more patches need to be checked. So instead, we used this method on each feature separately, meaning that instead of having a large bounding box, we would have several smaller ones. The set of feature pixels that we need to check for each patch. This reduced the time further to less than a tenth of a second which we were pleased with as it was around one percent of the prediction time. So we decided to use this pixelisation method for occlusion as it served the same job as occluding individual pixels but it also helped to remove some of the shape bias.



(a) Single feature with bounding box (b) Multiple features with single bounding box (c) Multiple features with multiple bounding boxes

Figure 3.14: Bounding box changes dramatically in size when multiple features are selected. The features are pixels in red and blue. Our solution was to use a bounding box for each feature (c) as the total area would almost always be smaller.

3.5 Metrics

We found initially that the manual occlusion method where the user occludes each feature one-at-a-time to be very time consuming as they would have to check each combination to see how the predictions changed. For n features, there would be $\mathcal{O}(2^n)$ combinations of features to try. Therefore, we gave the user the option to have their model to be analysed automatically with the click of the "Analyse" button. We found four metrics to be enough to summarise the analysis as they express four interesting properties of the particular model and image combination.

3.5.1 Largest Change of Confidence

An obvious first question people would have when presented with a tool like this is "what do I need to occlude in order to completely break the classification?". That is the question that the first metric tries to answer by finding the set of features to occlude that produces the largest change in the confidence value for the original class. This is done by looking at as many of the 2^n combinations of features as possible. This is by far the most time consuming of all the metrics to calculate though thankfully only forward passes of the network are required and so can be made faster significantly by using GPUs. We decided for testing to instead sample a constant number of feature combinations and pick the one that produces the largest change in confidence.

3.5.2 Most Important Feature

A related question would be "which of these features produces the most significant change?" This is similar to the previous metric although it only looks at a single feature. It is also used to sort the features when they appear in the UI. It makes sense that a user would be most interested in the most relevant feature first and since the features cannot really be named, it is best to list them in order of relevance.

3.5.3 Minimum Features to Keep for Prediction

Another interesting metric to look at is the minimum set of features that are needed in order to keep the original class as the top predicted class. This is similar to the concept of the smallest sufficient region (SSR) defined in [DG17].

3.5.4 Minimum Features to Occlude for Perturbation

The opposite to the previous metric is to find the minimum set off features that need to be *occluded* in order to change the top predicted class. This is parallel to the concept of the smallest destroying region (SDR) defined in [DG17].

Chapter 4

Feature Synthesis

After completing the feature occlusion tool, we decided to explore other approaches for explaining neural networks and their predictions. Before, we were looking at how parts of an image (features) make up a prediction and now we will explore how networks "see" these features using feature synthesis.

4.1 What is Feature Synthesis?

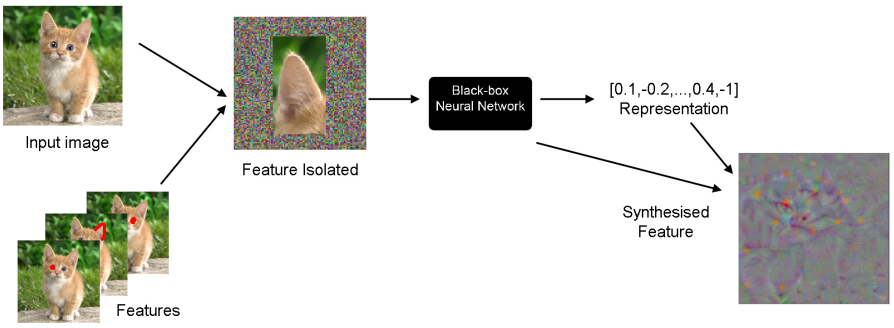


Figure 4.1: The entire pipeline for one example of the feature synthesis tool. First of all, the feature chosen to be synthesised is isolated from the rest of the image. This is done by using noise and centring the feature. Then, its representation is calculated for a particular layer. Finally, this representation is then used with the gradient descent algorithm to find a pre-image.

Feature occlusion is quite an interesting tool as it explains the effect that the important features have on the predictions that the neural network makes. We can cover up the eye of a cat, for example, and see the network not classify the correct breed of cat or we could occlude the basket of a hot-air balloon and the confidence of the network's prediction drops. This feature occlusion tool is used to essentially answer the question "what does my network do when I remove this feature?". Users who want to obtain more explanations from their neural networks may also be interested in other questions:

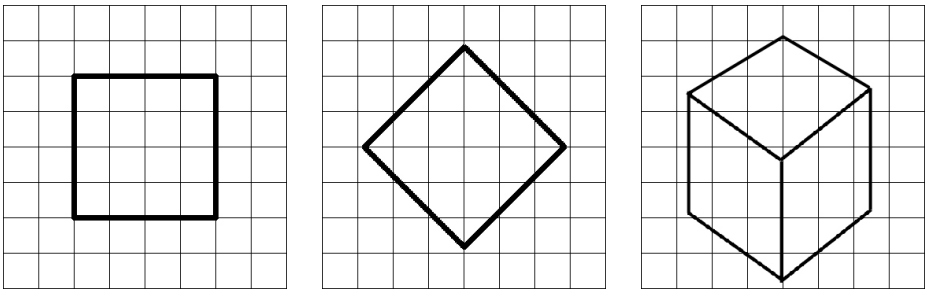
- How does my network "see" this particular feature?
- Would my network still be able to recognise this feature if it was rotated/scaled/translated, even in 3D space?
- Does the spacial position of this feature affect my network's prediction?

It is the first two of these questions that we are attempting to answer with the feature synthesis. The general idea for feature synthesis is to take a higher level representation for a particular feature (such as a deep layer in the network) and then try to generate an image based on this representation. The reason why this helps for answering the first two questions above is quite easy to explain. Since neural networks typically have to learn from millions of examples and thousands of classes while having a limited number of neurons, they must be able to compress the information

from these examples to these neurons. That means that they have to learn that multiple features from images can represent the same thing. On the very high level, this means that they can learn that multiple images of the same object from different angles and perspectives all represent the same object. It is reasonable to assume that this extends features that make up that object such as limbs, facial features and textures.

This does not only involved 2D transformations such as rotating, translating or scaling. As discussed in section 2.1, convolutional neural networks are designed to be invariant in affine image transformations. However, this does not take into account other transformations such as rotating on other axes than the one perpendicular to the image. It also does not take into account other slight changes such as elements of a feature be occluded or the shape is broken up. In order to be able to learn so many classes with a relatively small number of parameters, well-trained networks would have to be able to recognise the same feature in a wide variety of contexts and orientations. This usually means that a particular representation is the mean over this feature [Le13].

It is typically impossible to determine the elements that compose an aggregated statistic such as the mean without additional information. However, in the case with neural networks, we do have additional information: the network itself. Essentially, we can generate images that, when run through the network, obtain an approximation of the internal representation that we are interested in. This is the main idea behind the synthesis tool which takes three steps:



(a) (b) (c)

Figure 4.2: An example showing different types of rotation. **Left**: The original, front on image of a rotation. **Centre**: The same cube rotated 45 degrees on the axis coming out of the image. The original image can be clearly obtained by just rotating it back 45 degrees. **Right**: A view similar to an isometric view of the cube. In this case it is not possible to just transform the image to get the original. However, a lot of neural networks have to learn that all three of these images are similar.

1. User selects the feature they are interested in (obtained by the feature extraction method described in section 3.3)

2. The internal representation for the feature

3. The "pre-image" of this representation is then calculated

The pre-images of a representation will be different because you cannot invert non-linear functions like neural networks and also because these. Once you have a number of these pre-images, you can then see the range of images that the network recognises as the feature of interest. So these help to the answer the first two questions given at the beginning of this section which is the objective for this feature synthesis tool.

In the next two sections we will actually be looking at steps 2 and 3 in reverse order as that is the order that we ended up addressing them. The first looks at calculating a pre-image for a given representation for a neural network using a gradient descent algorithm. The second section looks at approaches we took to isolate the feature from the rest of the image and obtain its representation.

4.2 Synthesising the Whole Image

When it comes to feature synthesis, the first thing we tried was to synthesise the entire input image. This was done by computing a forward pass of the image through the network and then selecting a layer of the network to invert. It was mostly based off the work that Mahendran and Vedaldi have done that looks at calculating pre-images from a representation [MV15, MV16]. This basically involves starting with random noise as an input image $\boldsymbol{x}$ and minimising an objective function using gradient descent. The most important term for this objective function is the "loss" term that is the distance between the representation that we are trying to approximate Φ_0 and the representation of the network for the current input $\Phi(\boldsymbol{x})$. In this case, we used the normalised Euclidean distance:

$$\ell(\Phi(\boldsymbol{x}), \Phi_0) = \frac{\|\Phi(\boldsymbol{x}) - \Phi_0\|_2^2}{\|\Phi_0\|_2^2}$$

The representation is set of activation values for a single layer of the neural network. Though this function will naturally be non-convex as neural networks themselves have non-linearities, the authors assert that this will converge using stochastic gradient descent [MV15, MV16]. Their reasoning is intuitive rather than analytical. Since we already know that having a neural network learn is possible, and learning is intuitively more difficult than stochastic gradient descent in terms of the input then minimising a loss function is also possible.

4.2.1 Regularisers

Though this converges, this loss function is not enough. We ended up obtaining pre-images whose representations very closely approximate the representation but appear to be just noise. This is due to the very large space of possible values in what is essentially a $H \times W \times C$-dimensional space, where H, W and C are the height, width and number of channels of the input image respectively. In fact, this resembles the problem of adversarial examples that we discussed in section 2.3, where noisy images can have high confidence in a particular class. In order to obtain more natural looking images that better resemble the features that we are interested in, we have to add extra terms to the objective function in order to restrict the space of valid images. These terms are commonly known as regularisers.

Norm Regulariser

The first regulariser is used to keep the pixel values within a bounded range. Originally this was just the penalty $\|\boldsymbol{x}\|_\alpha^\alpha$ where α is a value that produces a high norm value e.g. 6 [MV15]. The only condition being that α must be even so the gradient is always in the correct direction. However, this term not ideal for colour images as the channels for a pixel are not considered as related. Therefore, this is modified to be isotropic in RGB space [MV16]:

$$\mathcal{R}_\alpha(\boldsymbol{x}) = \frac{1}{HWB^\alpha} \sum_{v=1}^{H} \sum_{u=1}^{W} \Big(\sum_{k=1}^{C} \boldsymbol{x}(v,u,k)^2 \Big)^{\frac{\alpha}{2}}$$

The term is normalised by the height and width of the image as well as the scalar constant B which is the average L^2 norm so that $\mathcal{R}_\alpha(\boldsymbol{x}) \approx 1$ though we found in practice it was more around $\frac{1}{8}$. The authors also suggest adding a hard constraint that clips the intensity of the pixels to around twice that of B which is defined in the paper as B_+. However, we did not find much difference in results with this additional constraint.

Total Variation (TV) Regulariser

The second regulariser that defined is much more interesting and is known as the total variation of the image. When used as a penalty term, it encourages the reconstructions to consist of constant valued patches. For discrete data like images, this is approximated with finite differences:

$$\mathcal{R}_{TV^\beta}(\boldsymbol{x}) = \frac{1}{HWV^\beta} \sum_{uvk} \Big((\boldsymbol{x}(v,u+1,k) - \boldsymbol{x}(v,u,k))^2 + (\boldsymbol{x}(v+1,u,k) - \boldsymbol{x}(v,u,k))^2 \Big)^{\frac{\beta}{2}}$$

Here $\beta = 1$ usually, but this leads to "spikes" in the reconstruction, as seen in Figure 4.3[MV15]. The sharpness of the image is sacrificed but these spikes are removed by setting the value of β to 2. It also made deriving the gradient a little easier as the outer power becomes equal to 1 which was convenient for us. The normalised value of $\mathcal{R}_{TV^\beta}(\boldsymbol{x})$ is also meant to be around unity as well. The authors suggest this so that parameters do not need to be finely tuned across different representation types. Fortunately, we were only using one representation type which is the last fully-connected layer of the network so we did not need to be as concerned.

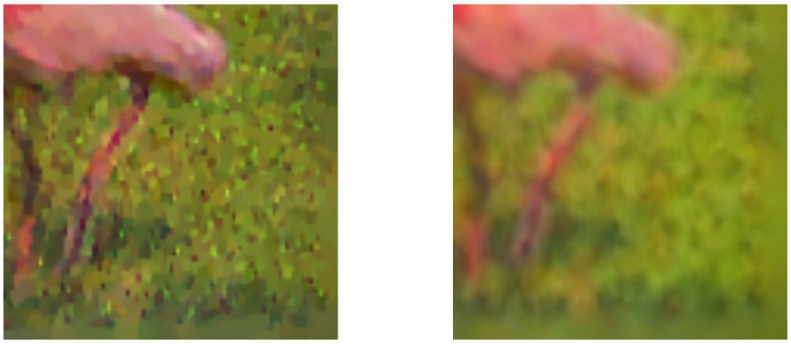


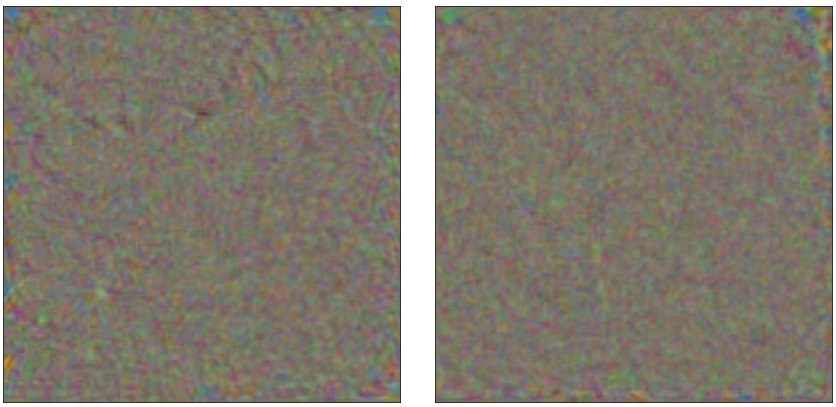
Figure 4.3: An example showing spike removal. **Left**: Reconstruction with spikes where $\beta = 1$. **Right**: Reconstruction where $\beta = 2$ where spikes are removed. Figure 2 of [MV15]

Jitter

Another regulariser suggested by the authors of [MV16] is the so-called "jitter" regulariser. This means that the image that is used to calculate the loss part of the objective function is shifted by a number of pixels on the x-axis and the y-axis with the amount shifted being different on each iteration. The resulting objective function now looks like this:

$$\mathcal{R}_\alpha(\boldsymbol{x}) + \mathcal{R}_{TV^\beta}(\boldsymbol{x}) + CE_\tau[\ell(\Phi(jitter(\boldsymbol{x};\tau)), \Phi_0)]$$

It is an expectation over τ as over a number of iterations, every available translation is tried. The maximum number of pixels that can the image can be shifted in any direction is defined as T which is a quarter of the stride for the particular layer. In our case for the VGG-16 [SZ15] network and the last layer fc8, this is $32/4 = 8$. The theory behind this regulariser is that it counterbalances the downsampling done by earlier layers of CNNs, leading to crisper images [MV16]. However, what we found was that it did not really help at all as it resulted in the loss being quite high (above 0.2) with no visible features.



(a) (b)

Figure 4.4: Two examples of jitter failing when trying to reconstruct an image of a cat. In the first, $T = 4$ and the loss is 0.224. In the second, $T = 8$ and the loss is 0.275. Neither of them show any features of a cat.

Original Image

The final regulariser that we considered was the distance between the original image x_0 and the current image:

$$\mathcal{R}_O(\boldsymbol{x}) = c\|\boldsymbol{x} - \boldsymbol{x}_0\|_2^2$$

where c is the constant multiplier for the term. This would "pull" the image towards the original image. However, there are a couple of issues with this regulariser. The first is to do with how we will isolate the feature from the rest of the image. If we use noise for example (which is the method we end up eventually using) the resulting synthesised image will be biased towards noise. The second and more important issue is to do with fidelity. The point of this tool is to synthesise what the network sees, not to replicate the image so using this regulariser would add and bias and not accurately reflect the representation.

The final objective function that we settled on was the original loss term with the norm regulariser and TV regulariser:

$$E(\boldsymbol{x}) = C\ell(\Phi(\boldsymbol{x}), \Phi_0) + \mathcal{R}_\alpha(\boldsymbol{x}) + \mathcal{R}_{TV^\beta}(\boldsymbol{x})$$

where C is a constant multiplier for the loss term which typically has the value of 1 for deeper layers. While implementing this gradient descent appears to be straightforward enough, we ran into many problems along the way, partly due to our own doing.

4.2.2 Implementation

Gradient Descent

The first part that we implemented, which was also the easiest was the gradient descent algorithm. This is where you want to find the value $\boldsymbol{x}$ (in this case an image) that produces as close to the minimum of a function $f(\boldsymbol{x})$ as possible. This is done by continuously subtracting a fraction of the gradient from a starting value for a number of iterations:

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta f'(\boldsymbol{x}_{t-1})$$

where η is the step-size or learning rate. We then also added momentum as well using the suggested value of 0.9 from the original papers [MV15, MV16]. This gives the algorithm short-term memory and allows it to converge faster as it oscillates less:

$$\begin{aligned} \mu_t &\leftarrow m \cdot \mu_{t-1} - \eta f'(\boldsymbol{x}_{t-1}) \\ x_t &\leftarrow x_{t-1} + \mu_t \end{aligned}$$

However, as we thought that we had gotten something wrong with the algorithm (which we will go into more detail later), we decided to adopt the full gradient descent algorithm outlined in [MV16]. In this case, it is a variant of AdaGrad. There is an individual learning rate for each component of $\boldsymbol{x}_t$ which are scaled by the inverse of the accumulated square gradient G_t. However, it is only in a small temporal window by using the momentum term m:

$$\begin{aligned} g_t &\leftarrow f'(\boldsymbol{x}_t) \\ G_t &\leftarrow m \cdot G_{t-1} + g_t^2 \\ \eta_t &\leftarrow \frac{1}{\frac{1}{\eta_0} + \sqrt{G_t}} \\ \mu_t &\leftarrow m \cdot \mu_{t-1} - \eta_t g_t \\ \boldsymbol{x}_{t+1} &\leftarrow \boldsymbol{x}_t + \mu_t \end{aligned}$$

where η_0 is the initial learning rate. To determine this value, what we want to look at is to reduce the value of $\mathcal{R}_\alpha(\boldsymbol{x})$ in a single iteration. This involves solving the equation $0 \approx \boldsymbol{x}_2 = \boldsymbol{x}_1 - \hat{\eta}_1 \nabla \mathcal{R}_\alpha(\boldsymbol{x_1})$. Assuming that all pixels in $\boldsymbol{x}_1$ have the intensity B so $0 = B - \hat{\eta}_1 \alpha / B$, so $\hat{\eta}_1 = B^2/\alpha$. We set the initial learning rate to one hundredth of this value. This was all implemented very easily using NumPy's element-wise computations.

Regularisers

The regularisers were also quite an easy part of the implementation to complete. Initially for the norm regulariser, we decided to use the original simple norm $\|\boldsymbol{x}\|_\alpha^\alpha$ whose gradient is simply $\alpha \boldsymbol{x}^{(\alpha-1)}$ (element-wise). When we thought that we were having problems with converging because of the regulariser terms (it was actually due to the loss gradient) we decided to switch to the isotropic version whose partial gradient w.r.t. each element $\boldsymbol{x}(v,u,k)$:

$$\frac{\partial \mathcal{R}_\alpha(\boldsymbol{x})}{\partial \boldsymbol{x}(v,u,k)} = \frac{1}{HWB^\alpha}\left(\alpha \sum_{k=1}^{C}(\boldsymbol{x}(v,u,k))^2\right)^{(\alpha/2-1)} \cdot \boldsymbol{x}(v,u,k)$$

This can be implemented very easily and computed efficiently using NumPy's element-wise operations and summations along dimensions.

The TV norm was a little more difficult to derive than the norm term but still quite straightforward. Since we were using $\beta = 2$ anyway, the outer power becomes 1 and so does not need to be considered. The important thing to realise is that any element $\boldsymbol{x}(v,u,k)$ only appears in a maximum of four different terms. The first two being those in the original definition of $\mathcal{R}_{TV^\beta}(\boldsymbol{x})$ where they make up the second half of each term. However, they also form the first half of two other terms: $(\boldsymbol{x}(v,u,k) - \boldsymbol{x}(v,u-1,k))$ and $(\boldsymbol{x}(v,u,k) - \boldsymbol{x}(v-1,u,k))$. Therefore the partial gradient w.r.t each element is:

$$\begin{aligned}\frac{\partial \mathcal{R}_{TV^\beta}(\boldsymbol{x})}{\partial \boldsymbol{x}(v,u,k)} = \frac{2}{HWV^\beta}((\boldsymbol{x}(v,u,k) - \boldsymbol{x}(v,u-1,k)) + (\boldsymbol{x}(v,u,k) - \boldsymbol{x}(v-1,u,k)) - (\boldsymbol{x}(v,u+1,k) - \boldsymbol{x}(v,u,k)) \\ + (\boldsymbol{x}(v+1,u,k) - \boldsymbol{x}(v,u,k)))\end{aligned}$$

Again, this is implement quite easily using element-wise operations and slicing to get the shifted pixel parts. An interesting thing we found is that for each of these, a row or column would be undefined. An example for the term $\boldsymbol{x}(v,u,k) - \boldsymbol{x}(v,u-1,k)$ would be where $u = 0$ as $\boldsymbol{x}(v,u-1,k)$ is undefined. We tried using the original element value although it did not make much of a difference so just leaving the value as 0 also worked.

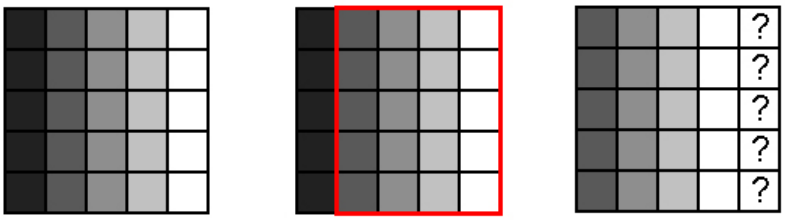
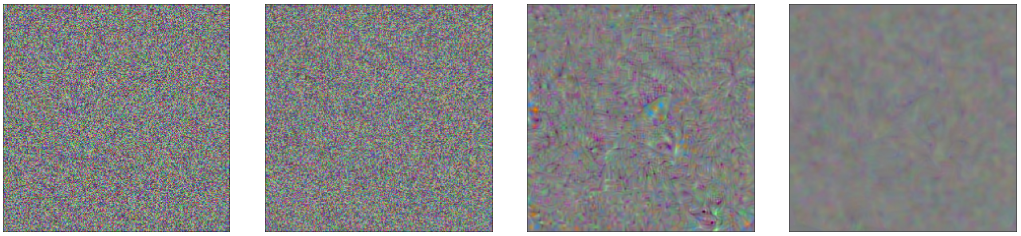


Figure 4.5: Illustration of the problem when calculating total variation TV. **Left**: The original image. **Centre**: Selecting the shifted pixel values which in this case are those for $\boldsymbol{x}(v,u+1,k)$. **Right**: Those pixels shifted to align with the original pixels. Now we have a column of unknown values.

Loss Term

Deriving the gradient of the loss term was the most time-consuming and difficult part of implementing this tool. This was mostly because of our original approach to the problem. We initially tried to do this manually. Our reasoning being that our implementation had to be machine learning library agnostic, so we were not dealing with specific library functions. Since we already had done this before when implementing LRP for the feature occlusion tool, we though this would be straightforward. Just using the chain-rule and backpropagating the initial gradient through all the layers until the input layer. The initial gradient being the the loss in terms of the representation:

$$\frac{\partial \ell(\Phi(\boldsymbol{x}), \Phi_0)}{\partial \Phi(\boldsymbol{x})} = \frac{\partial \|\Phi(\boldsymbol{x}) - \Phi_0\|_2^2 / \|\Phi_0\|_2^2}{\partial \Phi(\boldsymbol{x})} = \frac{2 \cdot (\Phi(\boldsymbol{x}) - \Phi_0)}{\|\Phi_0\|_2^2}$$



(a) (b) (c) (d)

Figure 4.6: The effect of the TV regulariser term on reconstructions for the cat image. In order, (a) shows no TV at all, (b) shows the result from a low TV multiplier ($V = B/2$), (c) from the multiplier we used ($V = B/6.5$) and (d) from a high multiplier ($V = B/15$). These were all from the same starting noise, which can be seen as the same part of the cat's head is synthesised in both (c) and (d).

We could then borrow some of the code we used to implement LRP but replacing relevances with gradients. We would only have to modify the backward and forward passes for the convolutional layers slightly as the biases are also used when calculating forward passes. Once we obtained the gradients for the input layer, then we just had to transpose them so that they would line up with the original dimensions of the image. Once we had this implemented and ran the algorithm for 400 iterations but we just ended up with very smooth noise with an average loss (the first term) reaching around 0.8 before rising again. We ended up trying various different things in order to see if we could get any better results:

- **Regularisers** The obvious things to try first were the regularisers. We tried increasing or decreasing their influence in the objective function. We also tried running them in isolation to make sure that the gradients were calculated properly which turned out to be the case. We found that no matter what, their values would be decreasing while the loss term would only decrease until it reached a particular value. We also experimented with adding the hard constraint to the norm regulariser but still found not much of a difference in results.

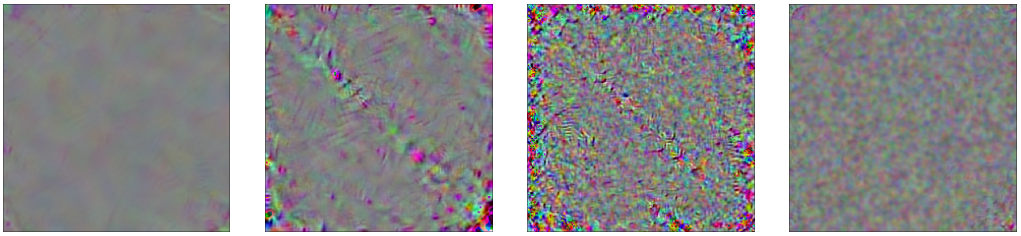
- **Learning rate** We initially started with the standard gradient descent which has a single learning rate. We tried various values to see if the algorithm was just diverging because it was too high. We also tried various learning rate decay strategies. In particular, we tried the adaptive learning rate which would double every time the loss decreased or stayed the same and would half every time the loss increased. What typically happened was that once the loss stopped decreasing, no change of the learning rate would allow it to start decreasing again.

- **Momentum** We also experimented with the momentum term, which was initially set to 0.9 as from the original papers [MV15, MV16]. We thought that this might be a problem as we found that after a few iterations, the gradients would not change must due to the momentum. However, even with a momentum of 0, the loss would still never get below a value of 0.8.

- **Initialisation** Another part of the algorithm that we experimented on was the initialisation. Initially, we just initialised the image with uniform random noise that is both positive and negatively valued as images are mean subtracted when fed into the network. We then tried using Gaussian noise instead and did not have much luck there either. Finally, we tried initialised the image with higher magnitude values. We found that the total objective function would decrease much faster because the gradients of the regularisers would be higher but still the loss term would not converge.

Some examples of these effects of these changes are shown in Figure 4.7.

What we had not realised was that the gradient for the loss term was incorrect. We initially realised this by only considering the loss term and the loss was still not converging. Since we thought that our implementation of the convolutional layers was wrong we tried the two simplest examples: the first two convolutional layers. These would have the least errors propagated backward and had the closest reconstruction to the original image. However, for one of them the



(a) (b) (c) (d)

Figure 4.7: Four failures initially when trying to synthesis an image with changes in parameters. The first reconstruction (a) shows a high total variation normalisation constant. The reconstruction (b)shows the effect of having a high value for C. We then switched to the AdaGrad gradient descent algorithm for (c). The final reconstruction was basically the same as (c) but the gradient of the loss term is normalised properly. Notice that some of them are quite noisy and none of them have any identifiable features.

reconstruction was transposed compared to the other. Our implementation for the backward pass had to be transposed. We got slightly better results when testing these convolution layers but the losses for deeper layers were still quite high. What we discovered was that the forward passes that we had manually implemented were actually incorrect as well, therefore our backward passes (which rely on the forward pass for determining how to backpropagate ReLU layers) were incorrect. We managed to fix these slightly by transpositions but we found that they were still incorrect by comparing analytic and numeric gradients. Numeric gradients are calculated by modifying $\boldsymbol{x}$ slightly and calculating the gradient like one would on a straight line:

$$\frac{d\Phi(\boldsymbol{x})}{d\boldsymbol{x}} \approx \frac{\Phi(\boldsymbol{x}+\epsilon) - \Phi(\boldsymbol{x}-\epsilon)}{2 \cdot \epsilon}$$

where ϵ is a small amount. We found that the difference between the two gradients was very large so we still had not calculated the gradient correctly.

Eventually, since we found that we were not going anywhere and had already wasted a couple of weeks, we decided to look at a specific library gradient calculation function. In this case, we looked at the Caffe [JSD+14] implementation as that was the most familiar to us of the framework. Fortunately, there already existed the function `backward()` with the Python interface. You would just need to set the `diffs` for the layer of the representation you were interested in to the initial change. For the case of the VGG-16 [SZ15] network this was the fc8 layer and the initial change was the difference between the target and calculated representations.

This was easy enough to use but when we tried it with everything else, the loss would still initially drop before then rising again. What we eventually discovered that this was due to a problem with transposing. For a particular image, the dimensions are $H \times W \times C$ in that order. However, it is initially transposed to what we thought was $C \times W \times H$ before being fed into the network. When we discovered that the gradients were in this initial shape, we thought we would just have to transpose the whole gradients matrix to get the original shape of our image. However, we discovered that it as actually transposed to $C \times H \times W$ but it was ambiguous. Fortunately after this fix, we actually had it working with quite impressive results as shown in the results in Figure 4.8.

4.3 Synthesising Individual Features

Now that we have found a way to invert an entire representation i.e. the representation of the network from a whole input image, the next step is to then try to invert only a single feature that the user has specified. What we initially thought of doing (as it was suggested in the original paper [MV16]) was to look at inverting a subset of the representation. While this could produce interesting results, the inverted representations would not properly reflect the features that they are trying to synthesise. This is due to two main reasons:

1. **Every pixel can influence every neuron** The obvious problem when trying to invert a subset of a representation to synthesise a subset of an image is the way neural networks are

Figure 4.8: Two examples of synthesis of a whole image. On the left of each is the image used to generate the representation and the right is a reconstruction. It is not as clear in the fox example but a snout can be seen towards the bottom right.

structured. For any neural network with some fully-connected layers, changing the value of any input can influence the value of any neuron in fully-connect layers. It is not guaranteed to happened due to the non-linearity of activation functions but it can happen. On the other side of this problem, pixels that we are not considering can also influence any neuron. Therefore, we would not only be concerned with finding neurons that are influenced by the feature we are interested but those that are only influenced by that feature. This problem is better illustrated like in Figure 4.9.

2. **Defining the correct subset** Related to the previous problem, how would one find the subset anyway? A possible approach would be to change the values of the pixels and observe which neurons change. However, due to the non-linear computations of neural networks, and the fact that any pixel can influence any neuron, this would be very difficult to do.

With these problems and the fact that it is not possible to mark the pixels we are not interested in as unknown, we decided to look at the approach we took in the previous chapter: editing all the pixels that we do not want to consider.

4.3.1 Using Noise

Our initial approach was to take our approach from the feature occlusion tool, but try the reverse by replacing pixels in the image that we were not interested in with random noise. The reasoning behind this is that we have to keep these pixels in anyway since we cannot set the values to unknown, therefore it makes sense instead to just avoid introducing bias. The only modification we made was to use the minimal bounding box of the feature instead of the individual pixels. The

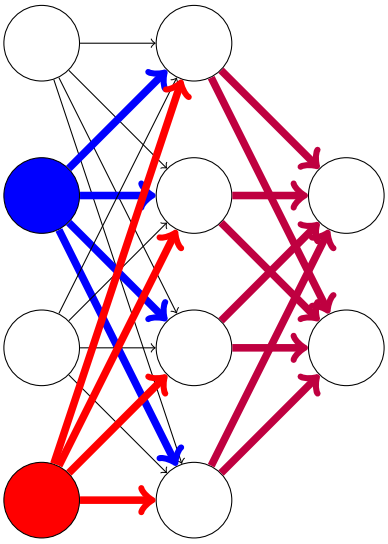
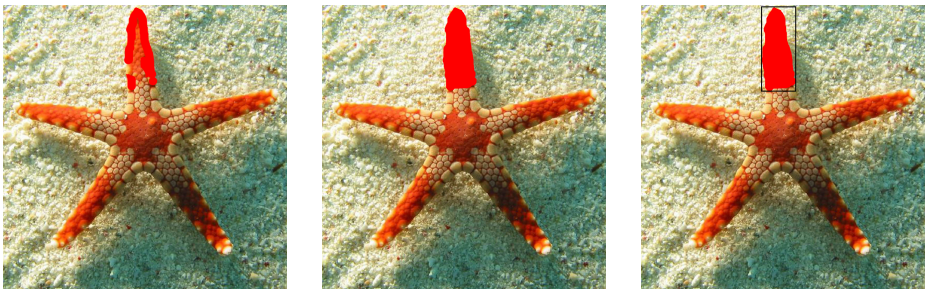


Figure 4.9: A toy example of a neural network showing that every input affects every neuron. The input marked in blue is an input that we are interested in while the input in red is an input that we are not interested in. Colour arrows indicate influence from these inputs and purple arrows indicate a mixture of both. As can be seen, every neuron whether we are interested in it or not can affect every neuron of a given hidden or output layer in a standard neural network. Therefore it is not possible to define the subset which are only affected by certain inputs.

main reason for this is because not all of the pixels of the feature are selected with the feature extraction tool. This is not a issue with the occlusion tool as even if not all the pixels are occluded, the shape of the feature is broken up when it is occluded. This does not work for feature synthesis as we want to synthesise the entire feature but this is solved by using a bounding box around the pixels so stray pixels that were ignored by the algorithm are kept. We found this technique quite successful with clear features being synthesised from multiple test images (see Figure 4.10).



(a) Pixels found through the feature extraction method in chapter 4.

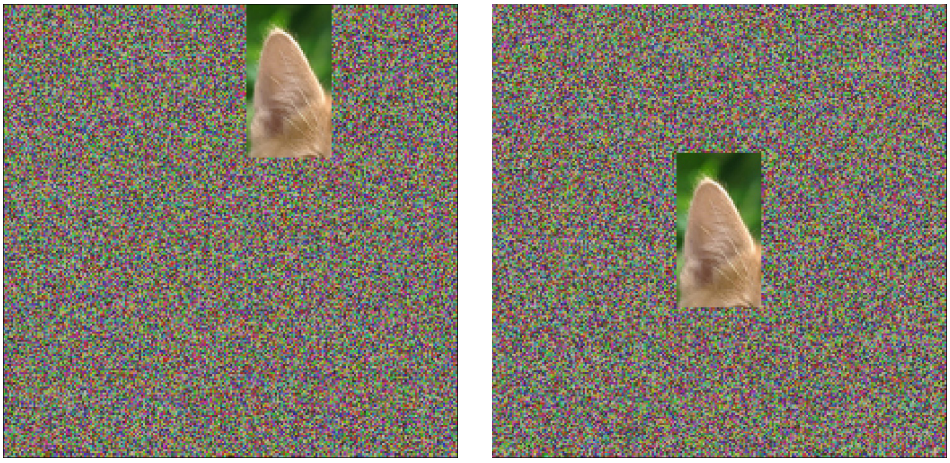
(b) Pixels that we are actually interested in for that particular feature.

(c) Pixels that we are interested in surrounded by the bounding box.

Figure 4.10: Examples showing why a bounding box is necessary. If a bounding box is not used, then a lot of pixels that we are interested in are ignored. In this case, we want to synthesise the entire arm of the starfish rather than the edge of it.

When we initially tried synthesising using noise we were not getting the best results, so we tried centring the feature in the image like in Figure 4.11 However, we found that there was not much of a difference when it came to the synthesised images. This makes a sense because, as we discussed in section 2.1, convolutional neural networks are designed in their architecture to be invariant in translation meaning that there should not be much of a difference if we translated the feature to the centre.

We obtained mixed results from this method The reconstructions have mostly noise probably



(a) Feature inplace. (b) Feature centred.

Figure 4.11: As well as using noise, we also tried centring the feature as well. This did not make much of a difference in reconstructions as convolutional neural networks are designed in their architecture to be invariant in terms of translation.

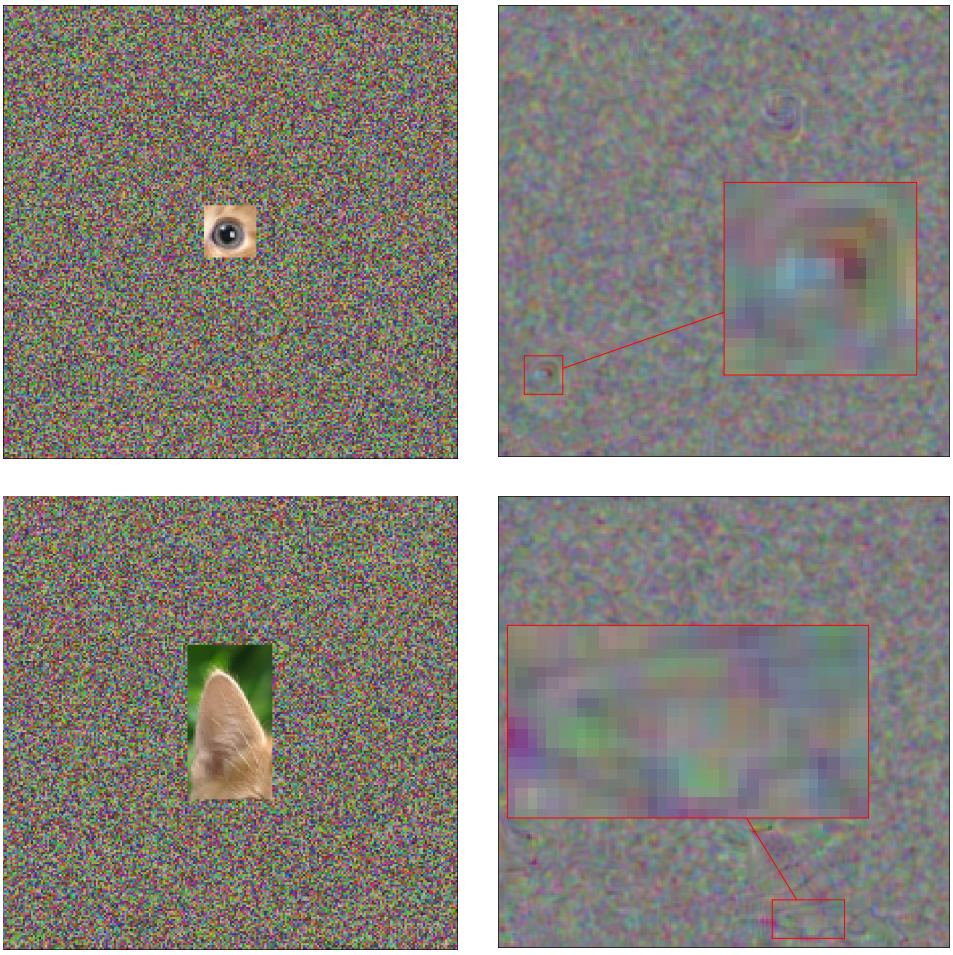


Figure 4.12: Two examples of synthesis of a feature using the VGG-16[?] network and the second last layer `fc7`. On the left of each is the image used to generate the representation and the right is a reconstruction.

because the source images are mostly noise and the reconstructed features are indicated with a red box. As can be seen, even for the same image, different features are synthesised to different qualities. The ear shape is only just about visible compared to the eye though we found that the

reconstruction error for the ear was higher than for the eye, 0.12 and 0.06 respectively. Even with changing the parameters we found that we could not lower it much further.

4.3.2 Resizing

Using noise was not the only way that we tried to isolate a particular feature from the rest of the image. We also looked at keeping only the feature itself as the input which can be done by resizing it. To do this, we used the `imresize` method from the SciPy[JOP+] library, which tries to preserve pixel values while resizing. That way, the non-determinism that is in the internal representation that is present from random noise is removed. We initially thought that this method would work better than using noise as less colour bias is introduced. However, we actually found that there were a few problems introduced when resizing features instead of occluding the rest of the area with noise:

1. **Aspect ratio** An obvious problem is that resizing to a fixed shape will almost always modify the aspect ratio of the original image. Since the, whereas the input shape for most neural networks, there always be at least some distortion. This means that the representation that the network will have of the particular feature will not accurately reflect the actual representation of the undistorted feature.

Figure 4.13: A demonstration of the effect that resizing can have on an oblong object such as a cat ear.

2. **Lower resolution** Another, though arguably not as critical, problem is with the change of resolution that comes with resizing an image. Depending on the method for resizing, there may be some errors that are added when it comes to enlarging an image. This is not as large of a problem, however, as the original image has the same amount of detail anyway as it is made up of a small amount of pixels.

3. **Removal of some context** A third problem that we found was some removal of context. An example we found was when trying to invert representations of the image of an eye. Even though the region of the image was a square so the aspect ratio was not changed, it still could not synthesise any features that remotely resemble an eye as seen in Figure 4.14. This is probably because the network itself is not trained to recognise facial features such as eyes so it is trying to invert a representation it perceives to be representing a particular class label. At least when using the noise method, the particular feature still appears to be a "part" of a class to the network instead of a class label by itself.

As well as some failure states, we also had other interesting cases. These would be where identifiable features were generated by the algorithm, typically eyes in the case of animals, even though they were not present in the input image. An example of this, showing the input image, is given in Figure 4.15. Despite the input image being of the nose and mouth of a cat, the interesting features that have been synthesised are eyes and eyes. We found the top five classes of the resized image to all be breeds of dog. We believe that the reason why these features were produced was because

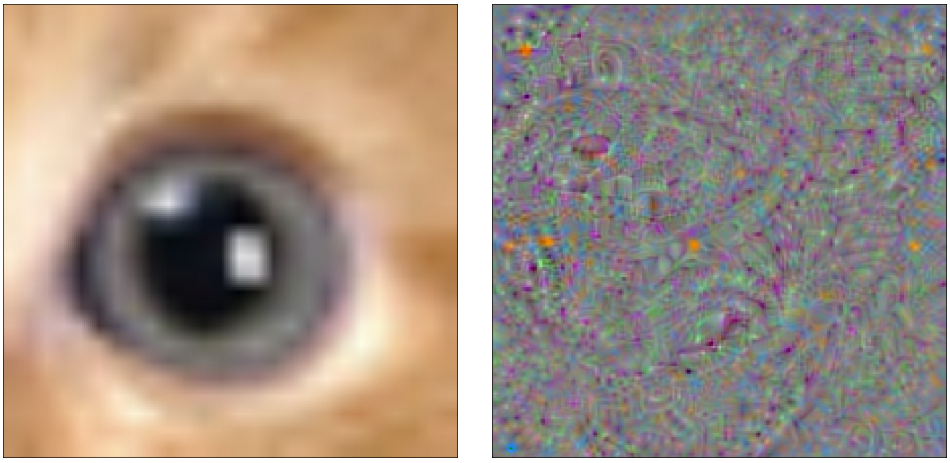


Figure 4.14: A failure case for resizing images. **Left**: The resized image of an eye used to generate the representation. **Right**: A reconstruction of the representation showing no sort of eye shapes.

the representation of the image in the network is closest to dogs. Therefore the network tries to reproduce significant features from dogs: their eyes and ears.

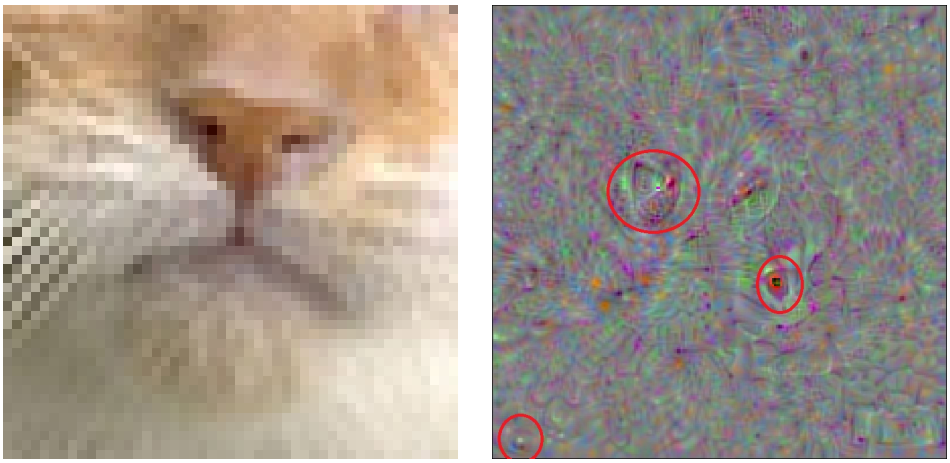


Figure 4.15: Example of a very interesting case when resizing. **Left**: The resized image of the nose and the mouth used to generate the representation. **Right**: a reconstruction from this representation with eye and ear shapes circled. Despite there being no eyes or ears at all in the resized image, these are generated by the algorithm.

Chapter 5

The Web Application

5.1 Overview

The main part of this project is an application that allows users to explore their neural networks using network explanation tools. The application would also be easily extendible for new tools in the future. We decided for this to be a web application mainly because it would be much easier to distribute to users as we would only need to share a url and they could use it immediately. It would also be much more portable for users with different operating systems and web browsers. The application itself is made up of three main parts:

1. **Front end** This is where the main user interaction takes place which is uploading and analysing neural network models. It is used to display the UI elements in the browser as well as to send commands to the back end. It should be fast and lightweight in its operations as well as having a lot of feedback and interactivity for the user.

2. **Back end** This is the part where all the main grunt work and algorithms are done. It does not need to be as fast as the front end in terms of interactivity but it should still be able to carry out computations efficiently. This is where all of the machine learning work is done as it is impractical to run it on the front end, especially in a language like JavaScript.

3. **Database/Storage** Since the model and image are impractically large to store in the database, we also had separate local storage for these files which we hosted locally on the same server as the back end. We planned to migrate them to an external host such as on Amazon's S3 storage platform as some point but we were more focussed on getting the explanation tools working correctly.

A detailed description of how these components interact with each other is also described in Figure 5.1.

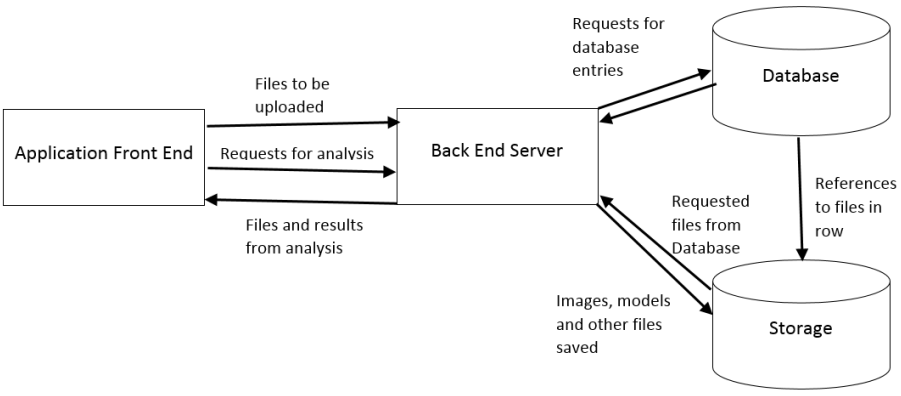


Figure 5.1: Basic architecture diagram showing how the four main components interact with each other.

5.2 Technologies

Since the web development ecosystem has expanded a lot over recent years, especially when it comes to areas like JavaScript libraries and frameworks, there are a wide variety of languages, libraries and frameworks to choose from. However, since this is a project related to deep learning, there were some technologies that were more set in stone than others. In this section we will briefly cover the technologies we have used to develop the application as well as the two network explanation tools. In the next sections, we will cover both front end and back end design as well as the database design. Finally, we will discuss two of the major design challenges that we had to deal with.

5.2.1 Python

Python is a dynamic programming language that can not only be used for scripting but general purpose applications. It is high-level and very flexible with a wide range of standard libraries. We decided to use Python for both our back end and for implementing the explanation tools for four main reasons:

1. **Easy to implement algorithms** Python was developed to encourage good coding style e.g. making indentation be part of the syntax. It has also been written to be high-level and very natural looking so it is much easier to implement algorithms from higher level representations such as pseudocode. Since our work is based off already existing algorithms. With libraries like NumPy, it also much easier to carry out operations that are essential in the field of deep learning such as matrix multiplications and element-wise operations compared to other languages.

2. **Support for machine learning libraries** From the above reason, a large number of machine learning libraries are either written in Python on have a Python interface. Therefore it should be much easier for us to extend our application to support networks written using other libraries.

3. **Plenty of existing libraries** One of the main strengths of Python are that there are already a large number of libraries for a wide range of applications. In particular, we found the SciPy[JOP+] and NumPy libraries to be useful for most of our computations such as matrix multiplication and they are used in other libraries like those for image processing.

4. **Portable with virtual environments** Through the use of virtual environments, Python applications can also be quite portable. If Python is already installed on another machine, you only need the virtual environment (or a list of libraries) in order to run the application on that new machine. This meant it was quite easy for us to deploy our application.

5.2.2 Django

Django is one of the main web frameworks for Python (along with Flask and Bottle) though it as a bit more flexible than the other two frameworks. It is much more suited to large applications compared to Flask, for example. We just needed a web framework for. We also found that the of apps (which can be composed into projects) to be useful for separating the different explanation tools. It would allow for fairly easy extension of the application to new explanation tools. There were also built in components such as database integration and user accounts that we found useful.

5.2.3 JavaScript

JavaScript is perhaps the primary language in front end development at the moment and is also being seen more and more in server-side applications. We mainly decided to use it due to its already widespread use for developing applications but also because we already had experience in developing user interfaces before.

5.2.4 React

React is one of the main JavaScript libraries for developing user interfaces and is maintained by teams at Facebook. It uses a composition instead of inheritance approach which can be initially

difficult to get used to. There were four main reasons why we chose React for developing the application's user interface:

1. **Wide variety of pre-built components** It's almost always best not to re-invent the wheel which is perhaps the main reason we chose React: there are already . There already exist a lot of both simple and complex components as packages hosted by the node package manager[1]. We also used Palantir's blueprint library[2] that provides a large variety of components such as forms, buttons and scroll areas.

2. **Easy to start prototyping** As well as having plenty of existing components, it also easy to get started developing a React application. Setting up a project only takes a single command and you can get started immediately. The JSX language which is like a HTML/JavaScript hybrid also makes this easy to do as you can easily mix HTML with JavaScript code.

3. **Previous experience** Unlike other popular JavaScript frameworks such as Angular.js or Vue.js, we already had previous experience developing using React so we already had a fair idea of how we would develop this particular application. It is also why we decided on a JavaScript+React front end as opposed to looking at developing the entire application using Python and Django.

4. **Portable** Like Python, React can be very portable when it comes to dependencies. Since the dependencies are also referenced in a separate file (package.json), only a single npm command needs to be run in order to download and install of the dependencies.

5.3 Front end Design

One of the requirements for this application is that it should be easy to use as it is intended for a wide variety of users. Therefore it needs to have a clear user interface which is both minimalistic (so the user is not too distracted) and also responsive and interactive. As mentioned in the previous section, this was written in JavaScript and React using Palantir's Blueprint library. It was designed as a single page application made up of many modules so it would be very responsive to the user's actions. A screenshot showing most of the application is given in Figure 5.2. As can be seen, the application is made up of many parts which we will summarise now:

1. **Image Collection** This is the main area where users can upload and select test images. The images displayed are only the ones that the logged in user had uploaded.

2. **Model Collection** Similar to the image collection, this is the main area were users can upload and select models to test. Only the user's models are visible.

3. **Tool Tab Bar** This is the bar of tabs used to switch between the various explanation tools, though there are only two at the moment.

4. **Tool Panel** This is the main area for the tools which covers the rest of the browser window. It switches to the appropriate tool when the user selects in the tab bar.

5. **Feedback Sidebar** This is for submitting general feedback on the application and opens an overlay for the user to do so. It's placed away from the rest of the application but is still visible.

Now we will dedicate a short section for each of the main UI components, describing our approach to the design and small challenges we had along the way.

5.3.1 Image Collection

The first UI component we ended up developing for this project was the image collection component. This allows the user to upload and manage their test images. It is essentially made up of only two parts: a scroll area where the images are displayed and a button for uploading images. In the application it takes up most of the height of the page. We decided on this because we needed

[1] https://npmjs.com
[2] http://blueprintjs.com/

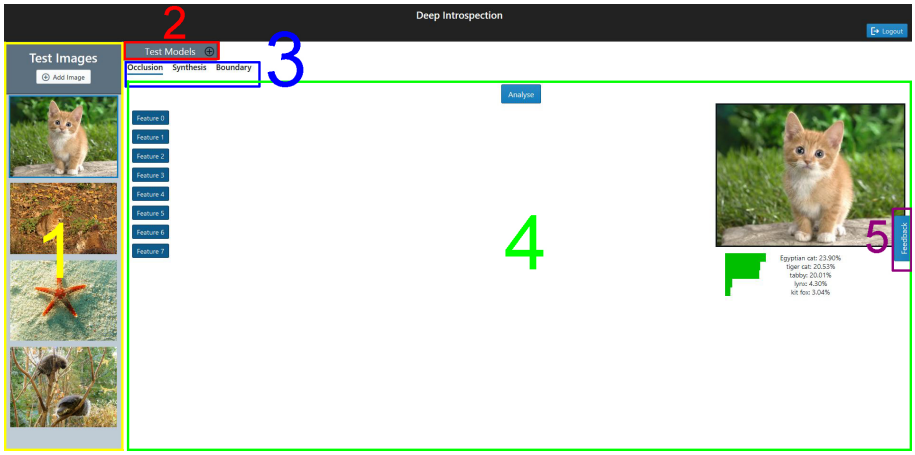


Figure 5.2: A screenshot of the main application with the five main areas labelled. Here the feature occlusion tool is shown.

it to be large enough to display the images clearly for the user. An alternative would be to maybe have a horizontal bar of images but we found the vertical bar to be both easier to implement and it would not use up as much valuable vertical space. The scroll area was achieved by using the `Scrollbars` component from the Blueprint library and it was very easy to implement.

Uploading Images

Uploading images are very simple from the user's perspective. They just need to click on the "Add Image" button and select the image that they want to upload. The image is then displayed alongside the other uploaded images. If the image already exists (by comparing data hashes), it is not uploaded and the user is told that it has already been uploaded. A file extension check is used to confirm that an image file has been uploaded and an error message is displayed instead.

Selecting Images

The user only uses one image at a time for analysis, so we wanted to make it clear which . We found this was simple enough to do by highlighting the selected image in the image collection with a blue border so it would be easy to spot. Selecting a different image is a simple as clicking on that image.

Toast

In order to be responsive to the user, we would have to provide feedback after an action happens so the user knows that something has happened. However, we also wanted to keep to the same page (otherwise everything would have to be reloaded again) so error pages were not an option. When uploading images, we decided to use notifications that are known as toasts. These are short messages that pop into existence and then vanish after a period of time. We decided to use Blueprint's Toaster system as it was simple enough to use but also had intents so colours could be use to distinguish between errors and warnings. A couple of examples of toasts for uploading images are shown in Figure 5.4. If the image had already been uploaded, an orange warning toast would appear and if the file was not an image a red error toast would appear instead.

5.3.2 Model Collection

This is similar to the image collection in that it is made up of a scrollable area for the models as well as having a button for uploading new models. There are, however, a few additions to this component. Most notable is that this component is collapsible. This is mostly due to the fact that this component was added towards the end of development. Most of the UI was already completed so there was no space to fit it without having to move around everything else. We considered putting it on the same column as the image collection but it would end up using too much space.

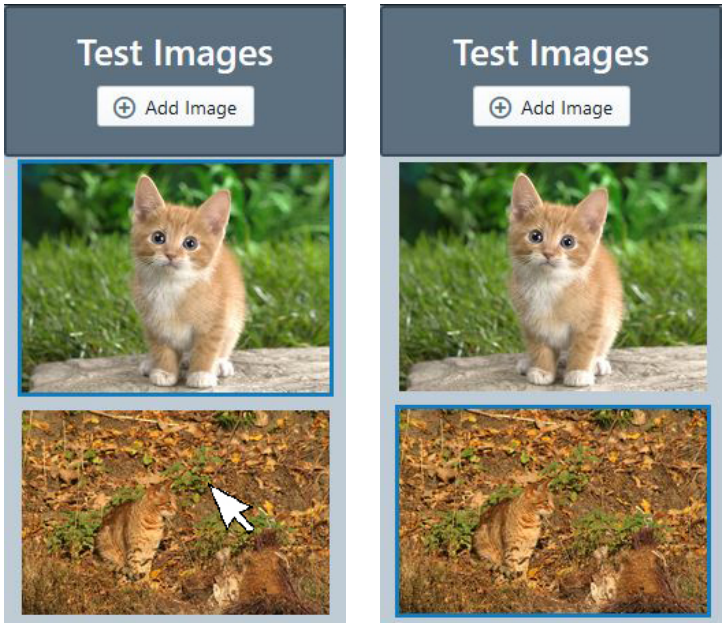


Figure 5.3: A sequence showing how selection works. Initially the first image of the cat is selected (indicated by the blue border) and then after clicking on the second image, it is then selected.

(a) (b)

Figure 5.4: Partial screenshots showing both the warning toast (a) and the error toast (b) for the image collection.

Therefore, we used the `Collapse` component from Blueprint so that by clicking the title bar. We thought this would work well as the user is not switching between models very often so the models that they can use do not always need to be visible.

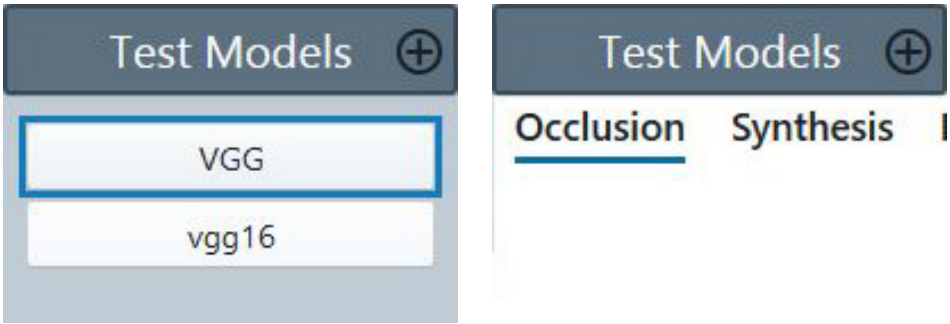


Figure 5.5: Partial screenshots showing that the model collection can be collapsed revealing the tool tab bar.

Uploading Models

When uploading a network model, there are a few files to upload so using a browser's default file selection (like for uploading images) does not work very well. Therefore we instead have our own custom file uploading UI element. This took the form of an overlay that would sit over the top of the application so we could separate it from the focus of the main application. Like for the image collection, we used toast notifications to indicate whether files were missing and also when all of the files had been uploaded. For invalid files, we decided to instead use red text on the

overlay itself as it would be more permanent compared to toast notifications.

Since the files for these models can be quite large and take a couple of minutes to upload,

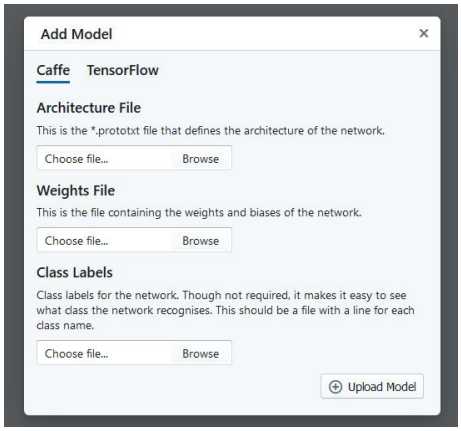


Figure 5.6: Partial screenshot showing the overlay displayed when uploading networks to the application. In this case, the form for Caffe networks is shown.

we needed some visible feedback to let the user know that the uploading was still in progress. In this case we used a progress bar component and since we split large files into chunks (more detail in section 5.6.1) we how far along the uploading was by how many chunks had been processed.

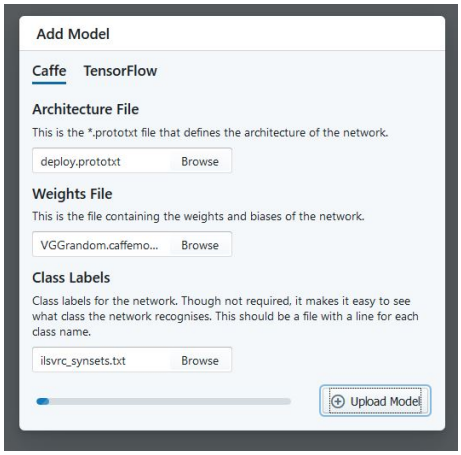


Figure 5.7: Partial screenshot showing the progress bar in the model uploading overlay.

Selecting Models

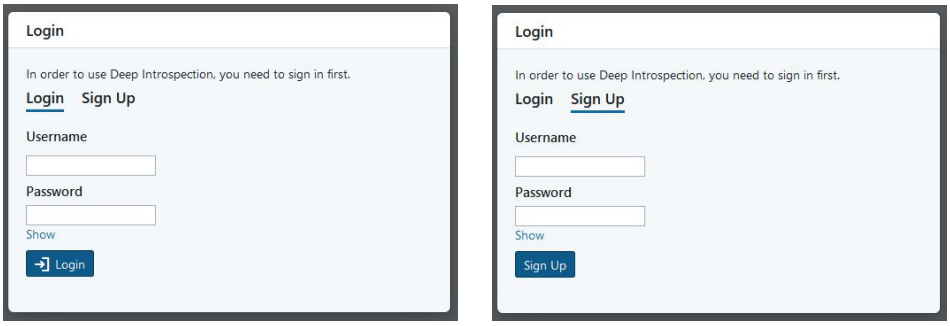
Similar to images, the user should be able to select which model that they currently want to test. We used the same approach for the model collection that we did for the image collection. The current model selected is highlighted with a blue border and it is easy to select another model by clicking on its name.

5.3.3 User Accounts

For the front end when it comes to logging in, the process is very simplistic. If the user is not logged in, they are presented with an overlay like the one in Figure 5.9. Again, we decided to use Blueprint Tabs, this time to separate signing up and logging in so the overlay would not take up too much space. We decided to only use the username and password fields as we did not need the user account system to be that complex. Session tokens are used so there is an automatic attempt to the log in the user when the application starts, meaning that the user is not inconvenienced by

Figure 5.8: A sequence showing how selection works. Initially the first model "VGG" is selected (indicated by the blue border) and then the model "vgg16" is selected.

having to login multiple times within a short period of time. We also added a simple logout button in the top-right corner of the page, so if the user wishes to switch accounts they can.



(a) (b)

Figure 5.9: Partial screenshots showing the two overlays for (a) logging in and (b) signing up.

5.3.4 Tool Area

The area for tools is simply composed of a Blueprint Tab bar and panels. To switch to a different tool, the user just needs to select the appropriate tab. We decided to have tabbed tools instead of separate pages for a couple of reasons. The first is that it is much more convenient for the user to switch tabs, but it is also faster a data needs to be fetched again if the application switches to a new page.

5.3.5 Occlusion Tool

The occlusion tool is made up of four main areas as illustrated in These are:

1. The list of features obtained by the feature extraction part of the occlusion tool
2. The area for automated occlusion analysis
3. The original image with the particular features occluded
4. The prediction results for that particular image

Since the user can occlude multiple features, we presented the features as a number of buttons. They are all initially set as activated so a user can then deactivate these buttons in order to occlude the corresponding features. When a feature is occluded, the set of occluded features are immediately sent to the back end so the pr

A problem with the feature extraction method is that there is no context to the features, so they cannot really be named and instead are numbered. For example, the user does not know what area of the image "Feature 1" is referring to. In order to remedy this, when a user hovers

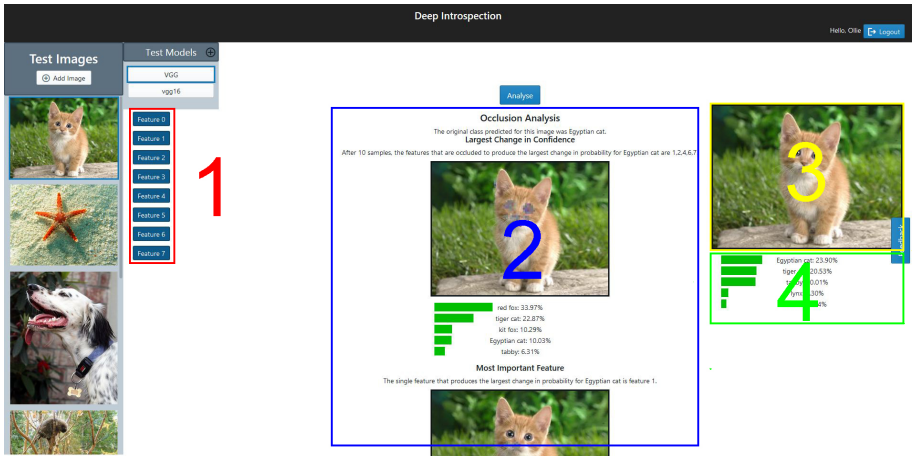


Figure 5.10: Partial screenshot showing the occlusion tool in action. The four main areas are labelled and the results from an automatic analysis are also shown.

over a particular feature, it is highlighted in red on the image so the user already knows what feature it is without testing by occluding it. An example of this is shown in Figure 5.11.

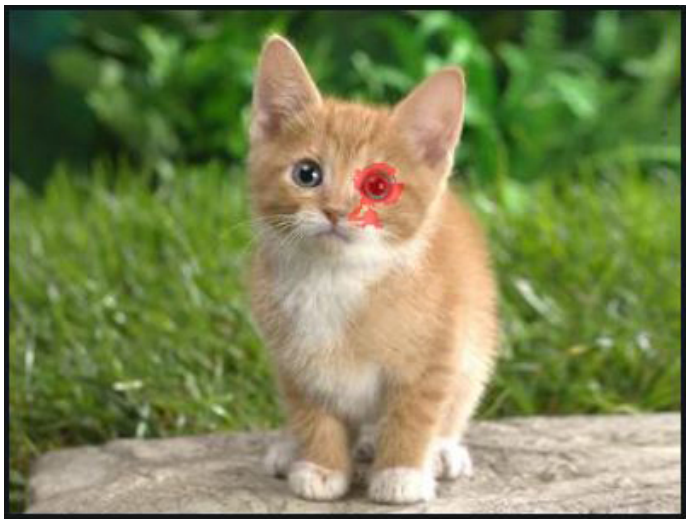


Figure 5.11: An example of one of the features been highlighted in red on the original input image.

5.3.6 Synthesis Tool

The synthesis tool is a lot simpler compared to the occlusion tool. We have the list of features as well as the area of the original image. However, we only have a simple grid gallery to display the synthesised images in the centre of the tool area. If a user wants to look at the images synthesised for a particular features, they just have to click on the particular feature. To generate a new image for that feature, they then have to click on the "Synthesise" button. Since this can take a significant amount of time, the user is warned with a tool-tip that this is the case but they can continue using other tools.

5.3.7 Feedback

For the feedback, these are displayed as overlays in order to save and draw the focus of the user away from the application itself. In order to not make the forms too intimidating, checkboxes, radio buttons and sliders are used for most questions as well as some text areas. There are two feedback forms:

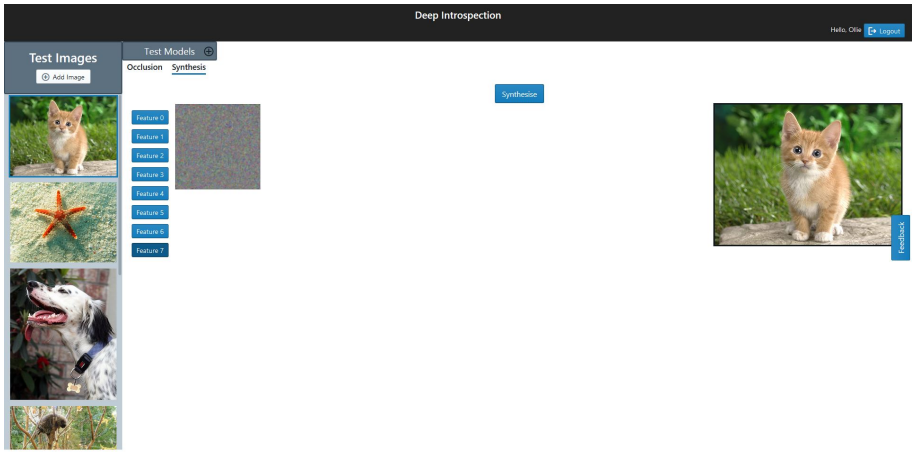


Figure 5.12: A screenshot of the synthesis tool with a single image for the feature synthesised.

1. A general feedback form that covers each of the tools and the application as a whole which can be accessed by clicking the button labelled "Feedback" on the right-hand side.
2. A feedback for the occlusion tool after analysis has occurred. This is linked to the specific analysis.

Multiple feedback forms were used in order to separate the feedback questions so they would not be too long. A toast notification is used to indicate whether the feedback has been submitted successfully.

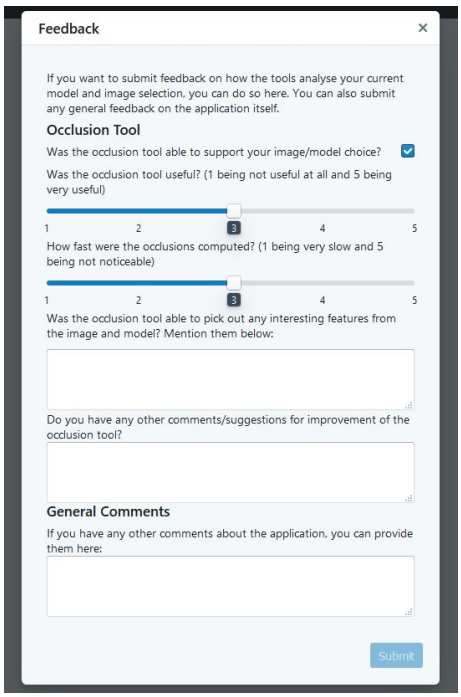


Figure 5.13: A partial screenshot of the overlay for the general feedback form. There is a mix of sliders and text areas for users to enter data.

5.4 Back end Design

Since most of the interesting design choices and implementation details are in chapters 4 and 5, this section will be relatively brief. However, we will discuss three interesting areas when it comes to the back end here that are not specific to the explanation tools.

1. **Django Apps** Django allows projects to be split into small, self-contained apps and so allowing greater modularity.

2. **Storage** How exactly will we store the data that is uploaded by users? Should we store results?

3. **URL paths** Since the back end acts mostly as an interface for the front end, we can give URL paths more context so that they are easier to use.

5.4.1 Django Apps

Django projects are composed of several smaller units called applications or apps. These are self-contained mini-projects that have their own settings as well as their own relative URL paths. We decided for the beginning to split our whole project into apps so that they would be easier to maintain and also so that they could be used in other projects. In the end we had six different apps that serve six different functions:

1. **Accounts** This is just used to provide an interface between the front end and the Django user accounts system. It allows users to create new accounts as well as login and logout.

2. **Evaluation** This is only used to allow the back end to store feedback from users.

3. **Features** This has both feature extraction (section 5.3) as well as the feature occlusion tool.

4. **Synthesis** This application is concerned with the synthesis tool as well as fetching synthesised images.

5. **UploadImage** This deals with uploading images as well as fetching images belonging to the user.

6. **UploadModel** This application deals with uploading neural network models as well as fetching them. Since the back end is the only part that deals with the models directly, only the names and IDs of the models need to be fetched for the front end.

The two tools were kept as separate apps, so they could easily be used in new applications and it also helps to keep their logic separated. It also made it more modular for when new explanations tools are added to the application.

5.4.2 Storage

It is impractical to store large objects such as images or model files in databases due to their largely varying sizes. Therefore, these are stored in the file system (`images` and `model` directories respectively) and linked to in the database entries. Generated files such as synthesised images as well as the lists of features are also stored in their own directories.

The reason why the lists of features are stored on the file system instead of calculated every time is for efficiency. It has been found that the algorithm to extract the features for a particular image and model takes up to two minutes. This may not initially seem like a long period of time but if this is done multiple times by multiple users it consumes a lot of resources. Since the results are deterministic anyway, we decided to store these features in a file so they could be fetched instead of generated if the user needs them.

5.4.3 URL Paths

Finally, another feature that Django provides is highly customisable routing and URL paths. For each app, you can set its path prefix to any valid string which allows great flexibility and intuitive interfaces for the front end. Django also allows for custom paths in methods as well as parameters in the path itself, which then appear as parameters in the corresponding method. This is very useful because it makes URLs more descriptive but it also makes it easier to separate methods from each other by having specific URLs. The parameters in the URL also makes the application RESTful and less effort is spent trying to process request bodies.

5.5 Database Design

In order to support users logging in and out as well as being able to switch between models and test images, some form of long form storage is needed. In this case it was a SQL database with SQLite being used. We initially tried using NoSQL databases with MongoDB[mon] but decided against it as we had difficulties integrating MongoDB with Django and we already had a lot of experience with SQL databases previously. Fortunately we did not need to define the tables directly as we used Django's QuerySet API. This is used as an interface for both creating SQL tables and also performing queries, not only making it easier for developers but also protecting against some security issues such as SQL injections. In the end, we had six different tables. They are as follows:

- **User** This the default user entity that comes with Django's authentication system found in `django.contrib.auth`. The fields that we are particularly interested in are **username** and **password**, the latter of which holds the hash of the user's password.
- **TestImage** This is the entity representing the images that the user uploads as test cases for their models. It has the **user** associated with it so only the images that the user has uploaded themselves are fetched. It also has a **hash** field which is only there to prevent duplicates of an image from being uploaded, saving storage space. The image files themselves are not stored in the table itself but under the local folder `images` and the attribute has an ImageField **image** that points to the file.
- **TestModel** Though most of the other data could be requested on the fly, due to the size of the files, this would be impractical for the network models so they would have to be stored on the server. For Caffe[JSD+14] models, an **architecture** and **weights** file are required. For TensorFlow[ABC+16, AAB+16] models, as will be explained in section 5.6.2, index and checkpoint files are also required but they do not need to be referred to in the model attributes. An optional **labels** file is also useful for a map between neurons and class labels.
- **FeatureSet** An entity used to store the sets of features generated by the algorithm described in section 3.3. Since each set of features is bound to a **model** and **image** there is a foreign key to the TestImage as well as the TestModel tables. The **features** themselves are stored as a data file in a `features` directory where each row is a feature represented as a list of tuples.
- **FeatureImage** Images created from the feature synthesis tool are also stored. Compared to the FeatureSet, they are not only associated with an **image** and a **model** but also a particular **feature** which is represented as just an integer. The resulting image (**feature_image**) is also an ImageField which points to the particular file which is stored locally under the `synthesised_images` directory.
- **Feedback** Used to store user feedback. The **feedback** itself is represented as a single JSON object as a string. That way feedback for different parts of the application can be represented with a single entity type. Each piece of feedback is associated with a particular **model**, **image** and **user**. It also has a particular **scope** which describes what tool the feedback is describing as well as a timestamp for when it was submitted.

An entity-relationship diagram showing how all of these are related is given in Figure 5.14.

5.6 Design Challenges

As always with large projects like this, we found some design challenges that we had to consider. The two particular challenges we had we to do with uploading large files to the back end server and extending our application from only supporting Caffe models to also supporting TensorFlow models.

5.6.1 Uploading Large Files

Initially a problem we noticed when trying to upload models and some images were error codes from the Django server. We found after some investigation that this was due to the maximum limit

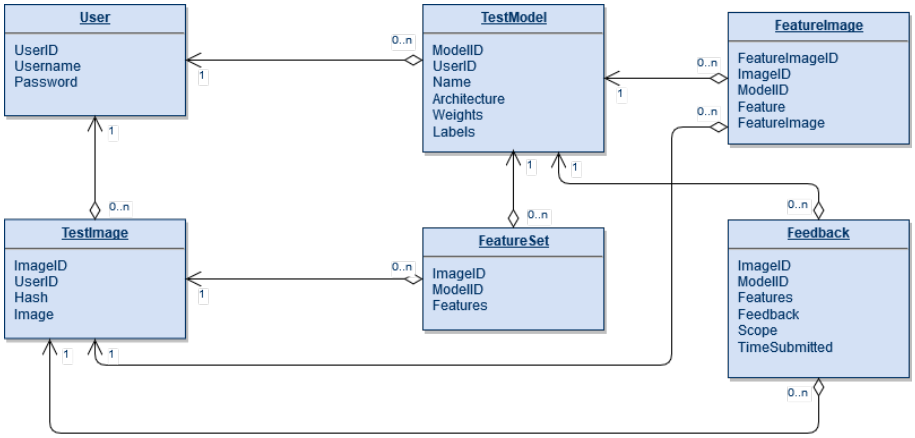


Figure 5.14: ER diagram of the entire database showing one-to-many relationships.

Django has for upload sizes. This is a setting can be changed though this would make the server more vulnerable to attacks as large files are allowed to be uploaded. So instead of trying to send files in one piece, we would split the files into 5MB chunks and send them across to be reassembled.

We decided to simply try to send them in order and once the last one was sent, send another request to reassemble file. However, since we do not know the order that the packets would be sent across the network, the request to reassemble could be sent before the. Therefore, we decided to use the responses from the server which would be unused to keep track of how many parts had been received using a counter. Only once all of the packets had been received and the counter reaches is the final request sent. This process is illustrated in the sequence diagram in Figure .

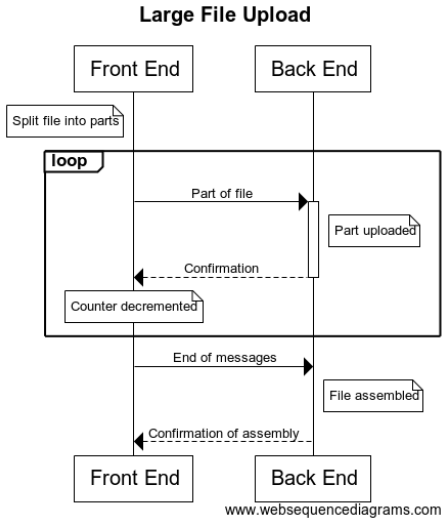


Figure 5.15: A UML sequence diagram showing how large files are sent from the front end to the back end.

Another issue that we found with sending the weights files was that the reassembled file had all of the parts but was invalid. Initially we thought this was an issue with the main assembly process e.g. data was being lost. However it turned out to be the way that the files were sorted. They were being sorted in lexicographical order. Since each part had a number following it, that would mean numbers with a different number of digits would not be sorted in the correct order e.g. a part ending in "50" would come after one ending in "401". To solve this we had to add a custom parameter that would sort numbers numerically and letter lexicographically.

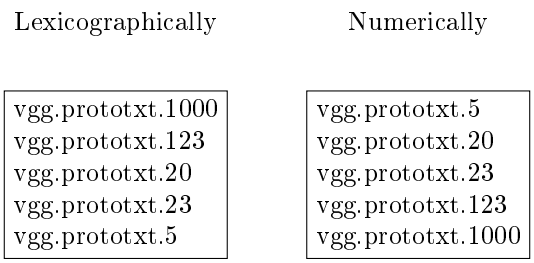


Figure 5.16: Example showing how a list of filenames would be ordered depending on whether they are ordered lexicographically or numerically.

5.6.2 TensorFlow support

Initially, the application as well as the explanation techniques could only support Caffe[JSD+14]. This id due to the fact that, as explained in background section 2.2, Caffe was used in most of the paper implementations and higher-level than a library like TensorFlow[ABC+16, AAB+16]. However, TensorFlow is the more modern and widely used of the two, not just in research but in industry as well. Therefore we decided that it would also be worthwhile to extend our application to support TensorFlow as well. To do this we had to extend two parts: uploading models and the back end for the tools.

Uploading Support

A problem with extending uploading to TensorFlow is that the number of files required is different. Other than the labels file, a Caffe network only requires two files: the architecture prototxt file and the weights file. TensorFlow, however, requires four files: a meta file (like the architecture file), a data file for the weights, an index file and a checkpoint file.

Initially, we considered having a separate table to separate the two models but there would be an issue. Since there would be two separate tables, there would be two separate model IDs so we could potentially have a user ID and model ID combination that would appear twice so they could not be individually specified. This could be remedied by having the front end also include the particular framework being used although we felt that it did not particular intent. Therefore, we decided that since the files Caffe models required were analogous to TensorFlow files and the checkpoint and index files do not need to be directly referenced, we did not need to change the table itself. TensorFlow and Caffe models could be told apart by the file extension of the architecture files (.prototxt and .meta). However, since the index and checkpoint files refer to file names that are user dependent, we decided to store each TensorFlow model in its own separate directory to prevent file name clashes and editing the files themselves. Since the two model types are uploaded differently, the back end functions for uploading had to be different. To solve this what we decided to do was add the particular framework name to the path so the front end would indicate which. This made sense as it was the front end where the user would choose the type of model that they were uploading.

Finally, we add to the UI so that the user could upload TensorFlow models. This was quite an easy part as we could basically take a copy of the original overlay for Caffe models and extend it so the user could upload index and checkpoint files. However, we did not want to add a separate button to the main UI for adding TensorFlow models and instead moved this behaviour into tabs which are similar to the tabs system for the tools. That way there would be a seamless transition if the user would want to upload a different type of network model.

Back end Support

Initially, since the algorithms were written using Caffe models, various Caffe methods were used directly in the code. Therefore, in order for these tools to support multiple frameworks, we had to make the networks being used framework agnostic. This turned out to be quite straight forward. Since the implementations and the attributes would be mostly different for the two frameworks,

we instead decided to follow Python's philosophy of "ask forgiveness and not permission". We defined two different classes `CaffeNet` and `TensorFlowNet` which had identical methods, even if their implementations were not the same. This was to avoid unnecessary bloat that defining a superclass would bring. Testing the two classes was simple as we just needed them to load the same network type (VGG-16 [SZ15] network in this case) and the same tests should return the same results. The only difference for the two would be when calling the constructor initialsly before passing the network to the particular algorithm.

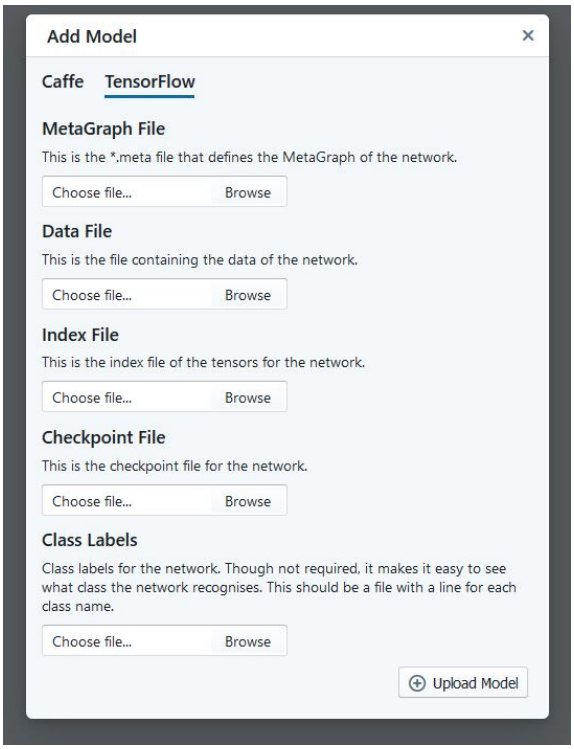


Figure 5.17: The upload model overlay showing the form under the TensorFlow tab. This is used to upload TensorFlow networks.

There was problem that we had when running through tests using TensorFlow models where a lot of memory was quickly used up. We found that this was due to a memory leakage from TensorFlow. Each `TensorFlowNet` stores a session internally whose contents are not freed from memory until the program ends. So for every network we were creating a new non-destroying TensorFlow session was created. Fortunately this was solved by just defining a destructor for `TensorFlowNet` that would clean up the resource.

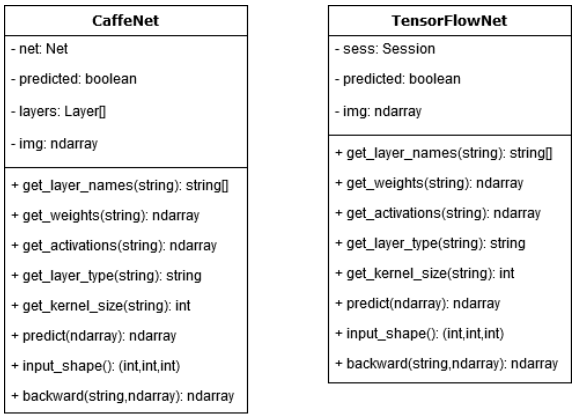


Figure 5.18: UML representations of the two networks. Notice that the methods are indentical in their signature and only the fields are different.

Chapter 6

Evaluation

6.1 Evaluation Model

Though the VGG-16 [SZ15] allows for impressive demonstrations visually, it is not the best model to run accurate evaluations. This mostly due to the fact that there are 1000 classes that need to be considered, meaning that testing each of these classes in order to evaluate the explanation tools is too time consuming. There is also the issue that the differences between these classes will not always be clear to a human and biases in the dataset can also play a role. Therefore, we decided to develop our own dataset and train a model that is as simple as possible while still being non-trivial. In this case, we decided to reduce the problem down to a two class problem which considers images of quadrilaterals and images of ellipses.

6.1.1 Dataset Generation

Since this is our own problem we need to obtain a dataset. We did this by generating our own dataset consisting of thousands of images of quadrilaterals and ellipses. In order to remove any biases, these were created by first generating a random background of greyscale noise, then overlaying the shape on top in pure white. Each shape was generated using `ImageDraw` from Python's PIL library and then randomly translated, rotated and scaled in order to produce a wide variety of examples for a given shape. This is to try and make sure that the only feature that discriminates between the two classes is whether the lines are straight or curved. We initially tried rotating the ellipse by rotating the two points used by `ImageDraw` to define the ellipse but this would only end up changing the shape. Therefore, we decided instead to scale and translate the ellipse and then rotate the generated image itself and this produced satisfactory results as shown in Figure 6.1.

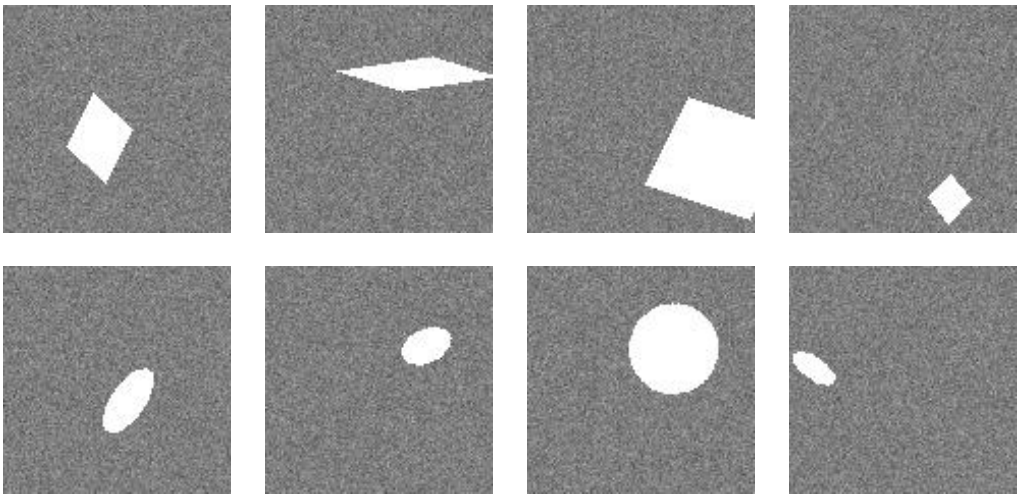


Figure 6.1: Four examples from the test dataset for each of the two classes. Notice that they all have different a different size, orientation and position.

We decided to make the images of size 112×112 in order to reduce the training time that the network would need but while also keeping clear and distinguishable shapes that the network could detect. It is also a factor of 224, the width and height of the input images for the VGG-16 [SZ15], whose architecture we used initially as the basis of our test network.

6.1.2 Network and Training

Now that we have the dataset generated, we now want to define and train a test neural network. We initially tried to implement this in Caffe[JSD+14] but we could not get the GPU version running on a machine, meaning that training would be very slow. However, we were able to install the GPU version of TensorFlow and so defined it using TensorFlow instead.

Initially we tried to use a stripped down version of VGG-16 by cutting down the number of convolution layers to up to the second pooling layer (two sets of two 3×3 convolutions followed by a pooling layer). However, the best loss that we got was around 0.5 before it would start rising again, regardless of the learning rate. So we decided to see if there were any other convolutional neural networks that were used for binary classification. We found the problem being that we were not using enough filters and large enough convolution kernels [Meh15]. We were only using 64 5×5 convolutions for the first layer while Mehri recommends at least 512 filters for the first layer. We decided to extend this network with a pooling layer of size 4×4 and stride 4 as well as another convolution layer with 1024 filters. This in then followed by a fully-connected layer with 2048 neurons and then a softmax layer with two neurons for the two classes. The network is illustrated in Figure 6.. After training for 40000 iterations we managed to achieve an accuracy on our validation set of 92.4%.

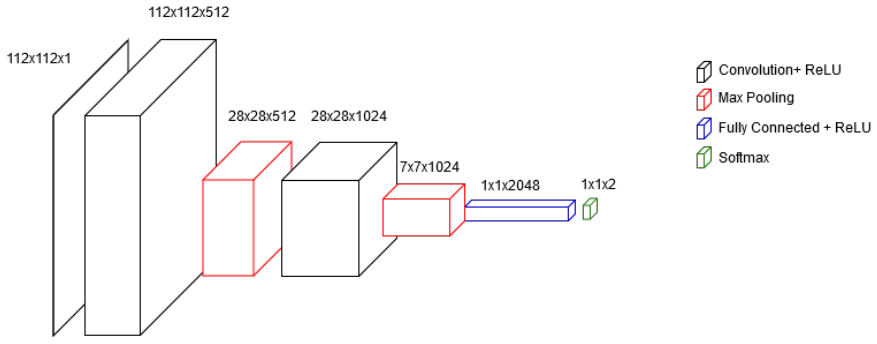


Figure 6.2: Architecture of our evaluation convolutional network.

6.2 Occlusion Tool

In order to evaluate our occlusion tool, the main part that we have to evaluate is the feature extraction method.

- Sensitivity Analysis (SA)[SWM17]
- CNN Fixations[MGB17]
- Layer-wise Relevance Propagation (LRP)[BBM+15]

Another method that we were considering evaluating against (and also possibly implementing in the tool) was the PatternAttribution[Kin] method. However, we found that it would be too slow to train the estimators required (over three hours on multiple GPUs) so would not be useful at all for our application.

6.2.1 Speed

One of the goals for the occlusion tool would be that our feature extraction tool should be reasonably fast. Not necessarily in real time but fast enough that the user can afford to wait. In order to

test the speed of each of the methods, we decided to run each of the methods on 20 examples and take the mean and standard deviation. Since these methods all involve an initial forward pass, we decided to calculate the mean of a forward pass and subtract that from all of the times. This test was run on a 4-core CPU with 8GB of RAM and no GPU calculations were used. The average times are given in Table 6.1.

As can be seen from the table, our method is the slowest of the four, being around twenty times

Table 6.1: Mean times for each of the four methods along with the 95% confidence intervals.

SA[SWM17]	CNN Fixations[MGB17]	LRP[BBM+15]	Ours
1.520 ± 0.168	0.936 ± 0.135	18.312 ± 1.309	18.454 ± 1.312

slower than calculating CNN fixations. However, CNN fixations is such a fast method because it only considers the activations of each layer while the three other methods require calculating gradients of some kind. Notice also that our method is only about a tenth of a second slower than LRP, meaning our extensions to LRP are relatively quick. Our method is still reasonably fast as the user will not spend a long amount of time waiting and the results can be saved to then be quickly fetched later. However, our main focus was not on the speed of our method but the accuracy, where our method definitely prevails compared to the others.

6.2.2 Accuracy

Evaluating the quality of our method is quite difficult to do quantitatively. What might appear a possible method of doing it is through measuring the localisation ability. One could identify the "ground truth" that the method should try and aim for (e.g. the object itself) then measure how much of the relevance is actually within that ground truth and how much of the ground truth is covered. However, the problem is trying to identify the ground truth. Since our methods are trying to identify areas that are relevant *to the model*, the "ground truth" of what is relevant depends on the quality of the network. The network may be biased towards the background, or the amount of light, or any other external factor that does not concern the object itself. Also, not the entirety of the object is necessarily relevant. For example, when trying to identify a road sign, do we always care about the colour of the writing? Therefore, we decided to evaluate our method most qualitatively by running each of the four methods on four examples and compare the results. These are shown in Figure 6.3.

For the CNN Fixations method, we show both the fixations and the resulting heatmap that is generated from these fixations. As can be seen from the examples, sensitivity analysis is by far the worst of the four methods as there are high values in areas of the image that are in no way related to the object. In fact, it is easier to spot the object by looking where the gradient values are low which is probably because the object itself is nearly all a constant white colour, therefore there is a very low gradient. CNN Fixations is better as it manages to localise the object in the image but does not appear to identify individual regions of the image judging by both the points and the heatmap. LRP gives improved results with not all of the features have a high relevance but there are . It can be clearly seen that our method is an improvement over LRP as it removes, and also removes small clusters that would not be very relevant. In particular for the quadrilateral examples, our method tends to remove the sides as their relevance values are below the threshold, leaving just the corners. The last thing to keep in mind is that our method also clusters these pixels into separate features which is necessary for our occlusion tool.

Though the method, on natural images it does not always work particularly well. Typically, more features would be found than necessary. An example would be the starfish image in Figure 6.4. Here, it is able to locate the object in question, but our feature extraction method has managed to split features into multiple small ones.

6.2.3 User Study

We also carried out a more holistic evaluation with both the occlusion tool and the application by having a small user study. We decided not to evaluate the synthesis tool for users as any synthesis

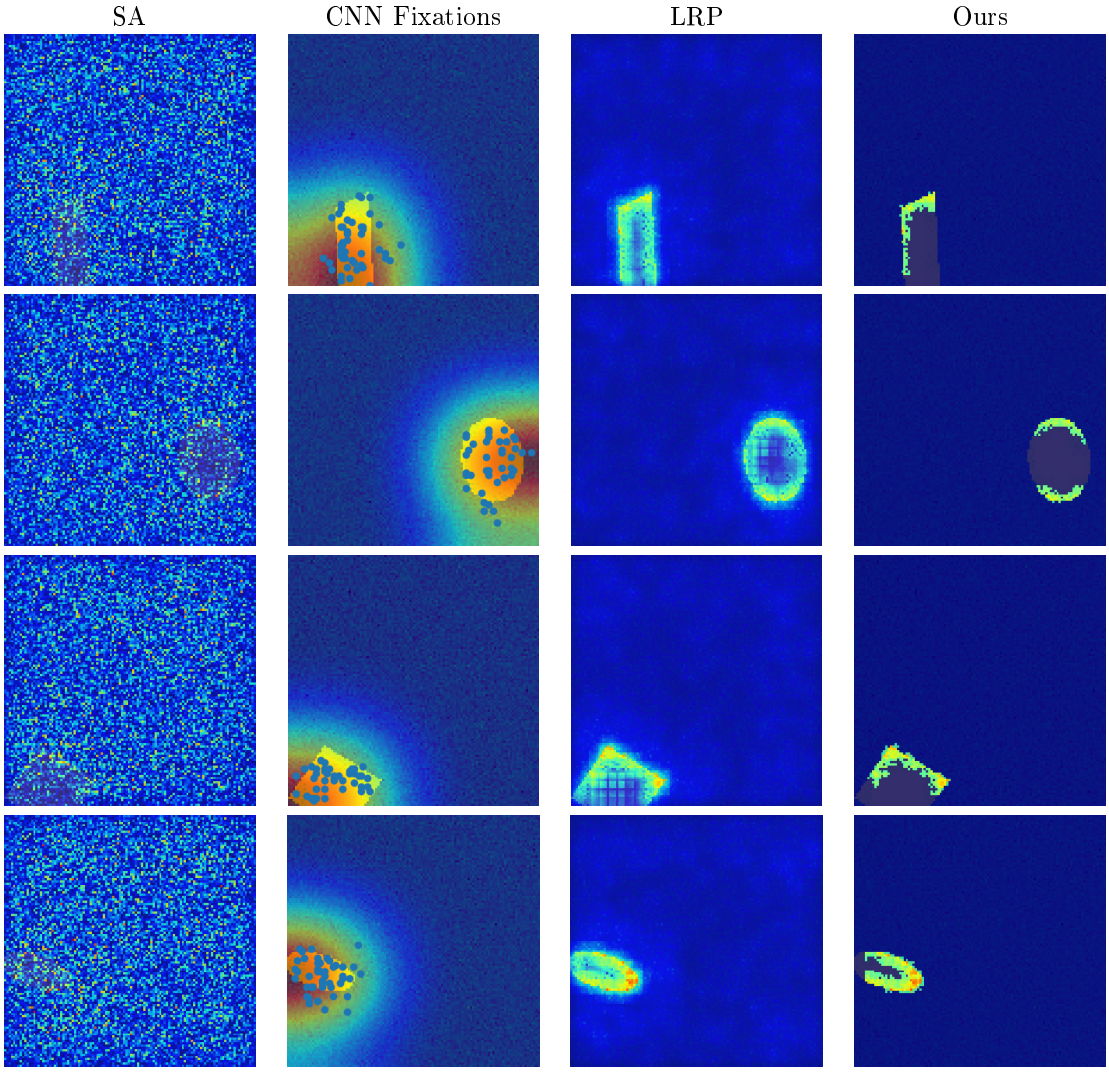


Figure 6.3: Comparison of the four methods. Sensitivity analysis first column is by far the most inaccurate while CNN fixations can localise quite well. Layer-wise relevance propagation is able to localise although it is quite noisy with small relevances found everywhere. Our method improves upon this by removing the noise and clustering the relevant pixels into higher-level features.s

would take far too long to be useful. We set up the application on a college internal machine with a VGG-16[SZ15] network and allowed various staff and postgraduates to try it and invited them to upload their own test images. We also gave them a basic description of what the tool is as well as how to use it as an overlay. We let it run for a few days to collect some feedback. The first issue we found while testing in the real world was when uploading images. Though we checked for whether a duplicate image is uploaded, we had forgotten to check if an image with the same filename was uploaded as all the filenames for the images we tested with were unique. This led to the original image being overwritten in storage so that the features found would only fit the original image. This was easily corrected by adding a unique ID to the filename.

Most of the feedback did not actually come from the built-in feedback we had added to the system but by email. The main piece of feedback we received was that we were not very clear about what the application was and how to use it. The description and instructions were considered unclear. This was most likely due to the fact that we had not written them with the layman in mind and assumed the user would know a lot about the problem. Some users were also surprised that the algorithm was not selecting human recognisable features as we had not made clear that the point of the application is to find features that the textitnetwork considers important. However, a lot of the users found the tool to be quite interesting and enjoyed experimenting with various test

Figure 6.4: An example where the feature extraction tool does not work as well as expected. An image of a starfish with all thirteen of its features. Notice that in the bottom right arm, what could be a single large feature has been split into four small features.

cases such as adversarial examples or examples where the network would not classify the image correctly. We changed the description and instructions to be a lot clearer and explain the purpose of the tool better and found we received a much more positive response.

6.3 Synthesis Tool

The other tool that we developed was the synthesis tool that synthesises individual features. It might seem that qualitative evaluation is the only worthwhile form for this kind of tool since we want to see how our synthesised . However, due to the way that these images are generated (from representations), there are also some quantitative methods from [MV16] that we can also use to evaluate the synthesised images. Since synthesising features of a particular is quite a novel method, we can really only compare it to the work of Mahendran and Vedaldi which it is based off. We will also look at qualitative evaluation by comparing features alongside their synthesised counterparts.

6.3.1 Reconstruction Error

The first part that we want to evaluate is how close our synthesised images are to the feature in terms of its representation i.e. the reconstruction error. In this case it is the euclidean distance. To do this we selected 5 sample images, ran our feature extraction tool on each of them, picked a random feature and synthesised it 10 times and took the mean for each of them. In this case the representation layer was the second last layer of the network. We found that the average reconstruction error was 1.67%, which is about a tenth of the normalised error found in [MV16]. This is a very promising result and we believe it is partly due to the fact that we are only trying to synthesis a part of the image. We found that we start with a lower synthesised error on average when using noise to occlude the rest of the image. For a smaller portion of the image we started at a smaller initial reconstruction error. This is possibly due to the fact that training convolutional networks learn to filter out noise as not being important so the noise used to occlude the rest of the image is treated the same as the initial noise.

6.3.2 Reconstruction Consistency

Since our goal is to be able to synthesise multiple images for the same feature, these should all converge to around the same small reconstruction error. We did this by looking at the standard deviation of 10 reconstructions of the same feature and averaging over 5 different features using the second last layer of the network. We found the mean standard deviation to be 8.8×10^{-4}, which is nearly proportional to the results found in Mahendran and Vedaldi. So all pre-images can be treated as easily good from the standpoint of the reconstruction error.

6.3.3 Human Identification

Finally, we also evaluated the synthesis tool qualitatively by seeing if the synthesised images could replicate features. This was an informal experiment which was done by choosing random features and comparing their reconstructions to the original. We have done this with four different examples of features and generated two reconstructions for each which can be seen in Figure 6.5. These were synthesised using the activations from the second last layer of the network, which is the fully-connected layer with 2048 neurons.

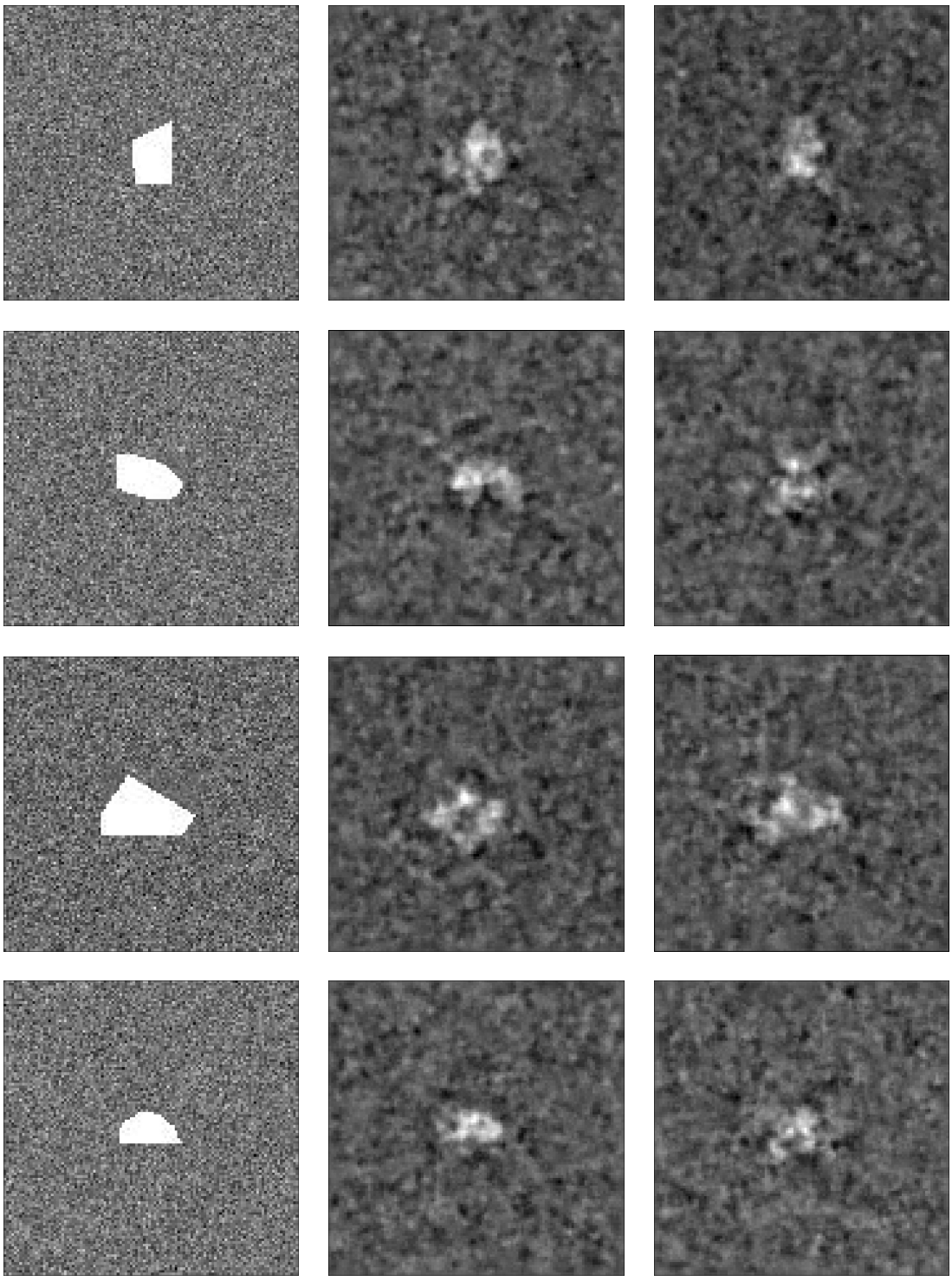


Figure 6.5: Four examples of features centred, each with two synthesised versions. Though the reconstructions are all quite noisy, it can be see that there is a more curved outline in the centre shape for the two features with curves compared to the two with only straight lines.

The main thing that we had to be careful of was how the feature may be clipped by the bounding box, producing straight lines. However, we found overall that this was not much of a problem. The reconstructed examples are quite noisy as can be seen. However, for the two features that have only straight lines (rows 1 and 3), it can be seen that the border of the white shape in the middle have straight lines while for the other two features, the lines are much more curved for the white shape in the middle. The reconstructions in the last row have less curves than those in the second row. We think that this is due to bounding box as discussed in section 4.3.1. creating more straight lines. What is quite interesting is that the main shapes in the features synthesised from the VGG-16[SZ15] network (Figure 4.12) have much less noise than those shown here. With a much more difficult classification problem (1000 classes on 224×224 colour images), the reconstructed images are much sharper despite the synthesising algorithm being the same. We posit a couple of reasons for this that could be explored further:

1. **More information to reconstruct from** The layer used for reconstructions for the VGG-16 network has twice as many neurons as the layer in our network, 4096 to 2048 respectively. This means that there is twice as much information that can be used to help reconstruct the image.

2. **Classifying images from ImageNet is a much more difficult problem**. The ILSVRC[RDS+15] involves a 1000-class problem. These classes can potentially involve examples that look very similar to each other where sometimes a human cannot even correctly classify them all the time. This means that in order for the network to be successful it needs to learn to recognise a lot of different shapes, patterns and colours. Our network deals with a much simpler problem that can be solved using manually written feature extractors to detect curved or straight lines with a near 100% success rate. This can mean that our network does not generate visually sharp images because it simply does not need to learn a lot of information.

One the main problems that we noticed with the synthesis tool is that it is not too flexible when it comes to the C multiplier which is used to control the loss term. It seems from Mahendran and Vedaldi that for the fully-connected layers of the network, C should have the value of 1 with as much regularisation as possible [MV15, MV16]. However, we found that a value between 5 and 10 actually worked better for us. It could be, though we are not certain without a wider variety of networks to test, that really the recommended value for C should get progressively smaller for each layer. In the VGG-16 network, the fully-connected layers are very deep down while for our own neural network they are relatively shallow (5th and 6th layer respectively).

Chapter 7

Conclusion

7.1 Final Thoughts

Finally, we now want to address the objective and challenges that we proposed at the beginning of this report as well as what we learned about this problem. The four objectives were:

1. An easy to use application
2. Expressive explanations
3. Human interpretable explanations
4. Flexible to architecture and implementation

As seen in Chapter 5, we have developed a full web application that allows users to run our two tools on their own networks and image examples. We have tried to make the application as simple as possible with as few steps as possible required and largely we believe that we have succeeded in this goal. Many of our users during the user study and have expressed, especially when we made the instructions a lot clearer in layman's terms. When it comes to expressive explanations, we believe that they are for most cases. The feature extraction tool is largely successful at finding salient features for a particular class. However, there are some failure cases where either salient features are ignored or are split into many small features unnecessarily. We believe that this is due to problems with our flood-fill algorithm and thresholds that could be easily tweaked in the future.

We also found our explanations to be quite interpretable to humans. This was, in part, related to the nature of our problem: classifying images. It is very easy for humans to process information with images, which is why charts and graphs are so popular for displaying information. The visual nature of the problem led to us looking at visual explanations, such as synthesised images and regions of the region. We don't believe our explanations would be as interpretable if we were looking at a different domain such as natural language processing or processing numeric data.

In terms of the flexibility of our application, it depends on whether the question is concerned with architecture or implementation. In terms of implementation, our network is (and will be in the future) flexible to different libraries. We decided initially to leave the TensorFlow integration as an extension but found it to be quite straightforward once we understood the TensorFlow API, taking us about a week to fully integrate. From this, it should be fairly easy to extend to other libraries, especially those like Keras that already use TensorFlow as part of their implementation. When it comes to network architectures, our application only supports basic convolutional neural networks. Our current implementation of LRP (used for the feature extraction algorithm) only supports convolutional layers and max pooling. There are many different layer types that it could be extended to such as inception layers[SLJ+15] and residual layers[HZRS16]. However, we were mostly concerned with getting it working on a complex example network as a proof of concept.

There were also three challenges that we had also given in our introduction which we thought that we would have problems with initially coming into the project. These were:

1. Algorithm performance
2. Human navigability
3. Human interpretability

Since we have already addressed interpretability when looking, we will only be discussing the other two challenges here. We initially thought that algorithm performance would be a problem due to the amount of data that needs to be processed in terms of network parameters. Algorithms like PatternAttribution [Kin], though provides a much more accurate explanation, was much too slow for our purposes. That's why we made the decision sacrificed some fidelity for increased performance. It was much more important to us that the application ran reasonably quick than for the explanations to be "perfect".

We think that we have addressed the human navigability challenge by mostly avoiding it in the first place. Initially we thought this would be quite a difficult problem due to our knowledge of the complexities of neural networks and we thought that we would have to display too much data. However, when we learned about the different explanation techniques, especially for image data, we found there were many that were quite easy to understand as they were presented in the form of images. This is why we focussed on visual explanations for this project instead so the user would not be overwhelmed with information.

We learned a lot throughout the course of this project. We were able to look at many different explanation techniques that have already been develop, though few have attempted to develop an application using these algorithms as explanation tools. We found that we could surpass a lot of the complexities of neural network explanations by finding interesting properties of the inputs and outputs rather than patterns in the networks themselves. We also learned that synthesising features can actually give quite satisfying results even for complex problems such as classifying natural images. Through the use of simple algorithms like gradient descent to produce explanations, we learned that though explaining neural networks is a very large problem and a very important one in deep learning research, it does not necessarily have to be a complex one.

7.2 Future Work

Unfortunately, due to the limited time frame of the project, we could not implement or fully explore every idea that we had for it. In this section we briefly discuss two possible extensions to the project that we thought would be interesting to pursue.

7.2.1 Boundary Images: Synthesising Between Classes

As well as the occlusion and synthesis tools that we have implemented, we also discussed the idea of a tool that would obtain "boundary images" between classes. What this means is, given two classes, try to generate an image that the network sees as in between those classes. In our case, the probability for each of those class neurons in the last layer would both be around 0.5. We could do this in a similar way to [MV15, MV16] and invert a representation to produce the image. There were two basic approaches that we thought of in order to obtain a suitable representation:

Raw Values

This is the very straightforward approach. For the last layer after the activation function, set the probabilities of the two neurons to 0.5 and the rest to 0. This can be done within the existing application easily although it may not completely reflect an accurate representation. Natural images will still have small values of probability for the remaining classes so it might be advisable to also add a small magnitude of noise to the remaining classes.

Mean Values

The other approach would be to select sample images for the two classes (the same number) and find the mean of their representations. This would lead to a more accurate representation as

it reflects multiple real world example. An issue that we could see from this in practical terms as it would not work with the current web application. This is because the application would require multiple examples of each class in order to work properly that the user themselves would have to upload.

Unfortunately, due to the time constraints of the project, we did not have the time to explore this idea further. However, we feel it would be an interesting tool to look at as it could help to define the turning point where a network's prediction might change from one class to the other. It could also help to identify which features that the network uses to discriminate between any two classes. This could be particularly useful if a user wants the network to explain how it could distinguish between two classes that are visually similar e.g. leopards and jaguars or different dog breeds.

7.2.2 Extending to Other Model Libraries

The obvious next step after adding new tools is to extend to the application to support more network libraries in order to support more users. After implementing both Caffe[JSD+14] and TensorFlow[AAB+16, ABC+16], the next choice could be Keras[C+15]. This is a high-level deep learning library (similar to Caffe) written in Python. What is easy about it is that the entire model (architecture and weights) is saved as a single HDF5 file making it very portable. There are a couple of approaches that could be taken to integrate Keras models.

1. **Create a separate `KerasNet` class with calls to Keras API** This is similar to what we did for TensorFlow and Caffe.

2. **Obtain the underlying network as a TensorFlow model and just use `TensorFlowNet`** Keras is really just a "model-level" library that provides building blocks of neural networks. It is not responsible for the low-level operations such as convolutions and tensor products. A low-level library called a backend is needed as a dependency to carry out those operations. TensorFlow is one of those backends supported so a TensorFlow graph is produced. That means we could build the Keras model in order to obtain the underlying TensorFlow model and create an instance of `TensorFlowNet` instead.

It could then be possible to extend the network to other libraries written in Python (or with a Python interface) such as Theano[BBB+10] and PyTorch[PGC+17].

Appendix A

User Guide

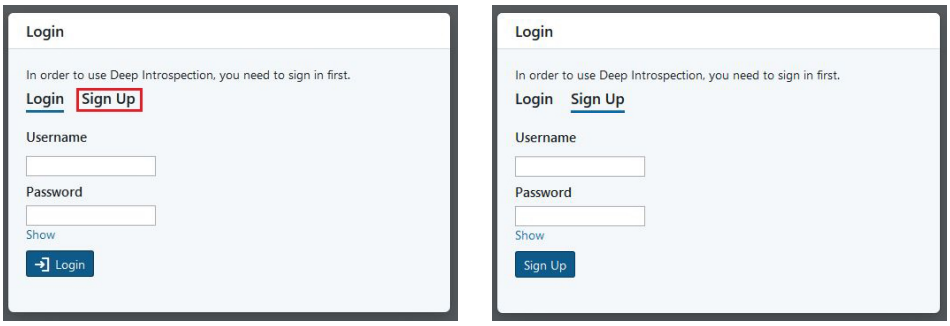
The Deep Introspection application is a tool that allows you to get neural networks to explain classifications in two main ways: feature occlusion and feature synthesis.

A.1 Initial Setup

Once you have accessed the application, you have to then create an account and upload test models and images.

A.1.1 Creating an Account

The first thing that you need to do is to create an account. When you first access the application, an overlay will be displayed asking you to login. To create a new account, click on the "Sign Up" tab and enter a username and password. If the username is unique the account will be created and you will be immediately logged in.



(a) (b)

Figure A.1: The login overlay for logging in and signing up. Click on the "Sign Up" tab highlighted to get the the option to create an account.

Your login information is saved but if you ever want to logout of the application you can do so by clicking the button labelled "Logout" in the top right-hand corner of the screen.

A.1.2 Uploading a Model

Uploading a model is a very straightforward process. Start by clicking on the plus button in the "Test Models" menu to get a dialogue box similar to the one below:

At the moment you can upload both Caffe and TensorFlow models by clicking on the respective tab. Caffe models require two files:

1. **Architecture** This is a .prototxt file that defines the structure of the network including its layers.

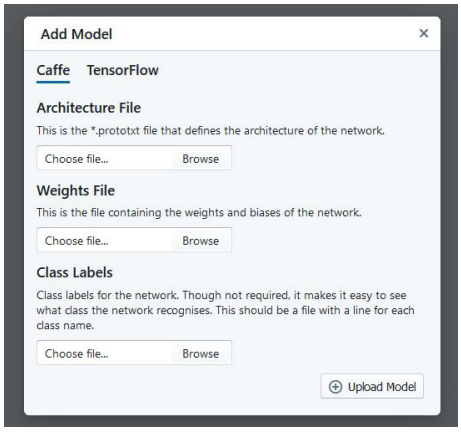


Figure A.2: The overlay used to upload models. This is currently on the Caffe panel but you can upload TensorFlow models by clicking on the TensorFlow tab.

2. **Weights** This is a .caffemodel file that defines the values of all of the network parameters i.e. the weight and biases.

TensorFlow models, on the other hand, require four files:

1. **Architecture** This is a .meta file that defines the execution graph of the network including its operations and tensors
2. **Weights** This is a .data file that defines the values of the network parameters.
3. **Index File** This is a .index file which contains metadata of each tensor in the network.
4. **Checkpoint** This is a file generated by TensorFlow that defines what current version of the network is required.

Though not strictly necessary, you should also upload a labels file so the application knows the names of the classes associated with the class neurons. This should be a text file with each line containing the name of the class that is associated with that neuron i.e. the first line should contain the first neuron and so on.

After you have selected the necessary files, click on the "Upload Model". You will be notified if there are any missing or invalid files. Depending on your internet connection speed and size of your model, this may take several minutes with a progress bar letting you know how far along the process is. Once it is complete your model will appear in the "Test Models" collection in the top left.

A.1.3 Uploading an Image

Uploading an image is an even more straightforward process. Just click on the "Add Image" button under "Test Images" and you will be presented a similar window to the one below for you to browse for an image.

Once you have selected an image, it should appear in the collection of test images on the left-hand side of the window.

Now that you have set up an account and uploaded a test model and some images, you can now explore the two explanation tools provided. The first being the occlusion tool.

A.2 Occlusion Tool

Before using the tool, you need to select the model and image that you were . To clicking on the model name in the "Test Models" panel and the image in the "Test Images" column. You can either occlude features manually or perform an automatic analysis.

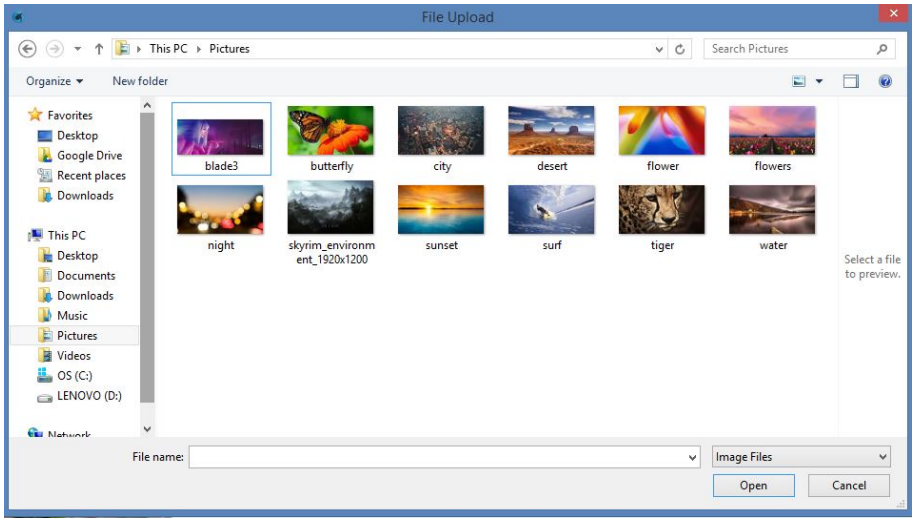


Figure A.3: An example of a window used to browse for images to upload.

A.2.1 Occluding Features

On the right-hand side of the tool is the test image and any features that are currently occluded as well as the top five classes for that image. On the left-hand side is a list of all of the features. Each of these is a toggle to occlude or keep that feature. Light blue toggles are those whose features are currently occluded. Though the features are numbered, you can see where they refer to in the image by hovering over them and they will be highlighted in red.

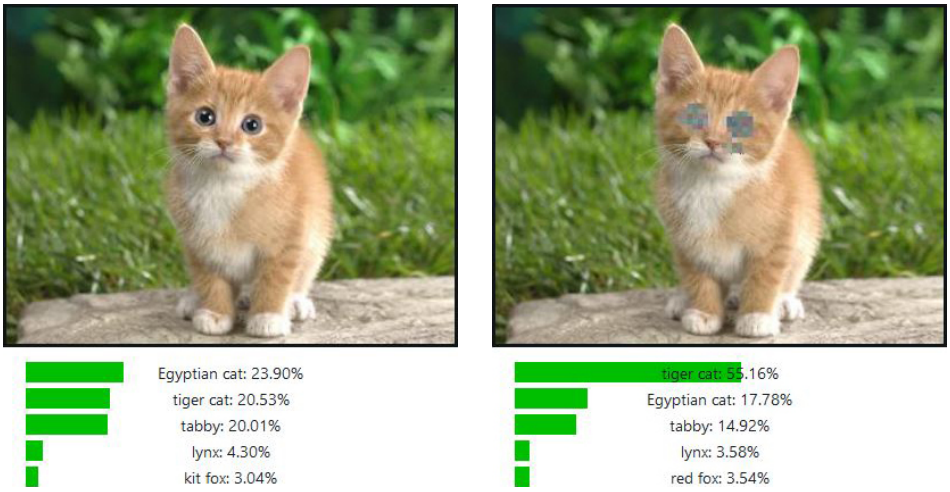
Figure A.4: An example of a feature, in this case the eye of a cat, being highlighted.

When you click on one of these buttons, the feature is then occluded or put back into the image and the top five classes change like in the example below. You can occlude multiple features at any one time.

A.2.2 Analysis

You can also run automatic analysis by clicking on the large "Analyse" button highlighted below. After between 20 and 30 seconds, you should get a result like the one below which displays four metrics:

1. Features required for largest change
2. Most important feature
3. Minimum features required to keep classification
4. Minimum features to occlude for a change in classification



(a) (b)

Figure A.5: An example showing how the top five classes change when multiple features are occluded. In this case, it is both eyes of the cat that are occluded.

For each metric, the image with those features occluded is also included.

This is the first explanation tool that occludes features. The second tool looks at synthesising them.

A.3 Synthesis Tool

It can be accessed by selecting the "Synthesis" tab next to the "Occlusion" tab near the top of the window. It is even simpler than the occlusion tool as can be seen below:

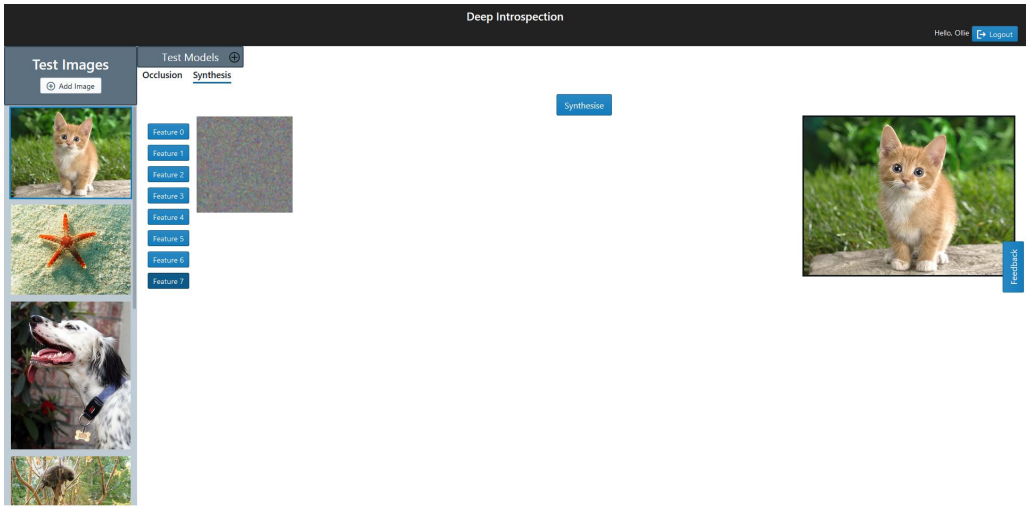


Figure A.6: An example of the synthesis tool for the cat example. So far only a single image has been synthesised for this particular feature.

Selecting a feature involves clicking on the button associated with it. Any synthesised images that you have already generated for that feature will be displayed. To synthesise a particular feature, click on "Synthesise". After some amount of time, depending on your model, a new synthesised image will be displayed. You can synthesise a particular feature as many times as you wish to. To get a closer look at the synthesised image, click on its thumbnail and the image will be displayed in a lightbox.

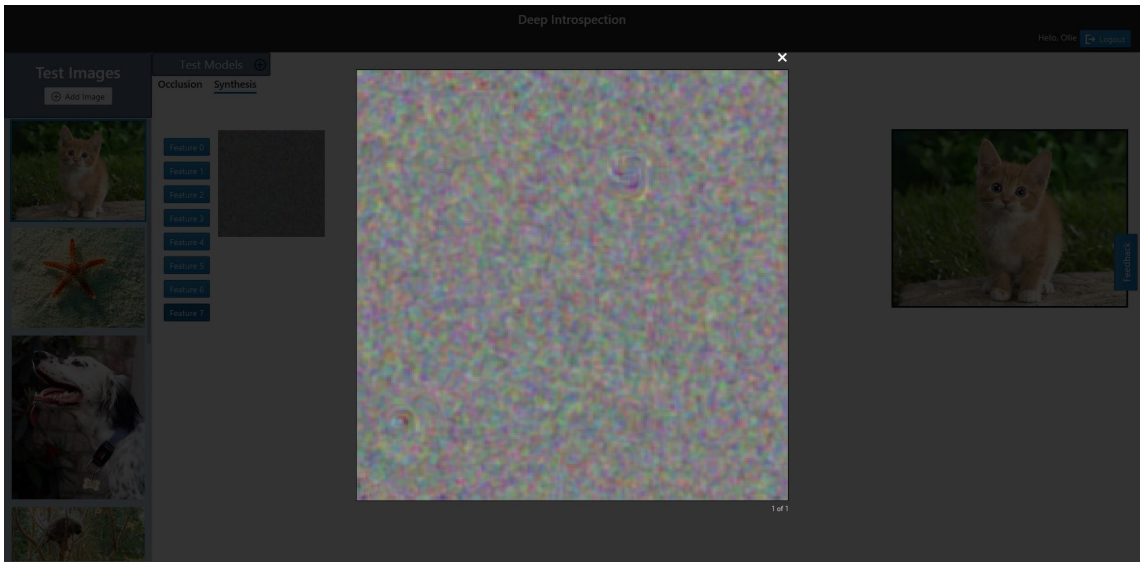


Figure A.7: When a synthesised image is selected, it is displayed in a lightbox.

Bibliography

[20117] Tutorial: Implementing layer-wise relevance propagation. http://www.heatmapping.org/tutorial/, 2017.

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[Bar17] Shane Barratt. Interpnet: Neural introspection for interpretable deep learning. 10 2017.

[BBB+10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, 2010.

[BBM+15] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[BHC15] Vijay Badrinarayanan, Ankur Handa, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling. *arXiv preprint arXiv:1505.07293*, 2015.

[C+15] François Chollet et al. Keras, 2015.

[CLG+15] Rich Caruana, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1721–1730. ACM, 2015.

[DAHG+15] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

[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 *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[Des16] Adit Deshpande. A beginner's guide to understanding convolutional neural networks. https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/, 2016. [Accessed 9 January 2018].

[DG17] Piotr Dabkowski and Yarin Gal. Real time image saliency for black box classifiers. *arXiv preprint arXiv:1705.07857*, 2017.

[DLW14] Jifeng Dai, Yang Lu, and Ying-Nian Wu. Generative modeling of convolutional neural networks. *arXiv preprint arXiv:1412.6296*, 2014.

[DSZB17] Yinpeng Dong, Hang Su, Jun Zhu, and Fan Bao. Towards interpretable deep neural networks by leveraging adversarial examples. *arXiv preprint arXiv:1708.05493*, 2017.

[DSZZ17] Yinpeng Dong, Hang Su, Jun Zhu, and Bo Zhang. Improving interpretability of deep neural networks with semantic information. *arXiv preprint arXiv:1703.04096*, 2017.

[DTZ17] Dawei Dai, Weimin Tan, and Hong Zhan. Understanding the feedforward artificial neural network model from the perspective of network flow. *arXiv preprint arXiv:1704.08068*, 2017.

[FH17] Nicholas Frosst and Geoffrey Hinton. Distilling a neural network into a soft decision tree. *arXiv preprint arXiv:1711.09784*, 2017.

[GF16] Bryce Goodman and Seth Flaxman. Eu regulations on algorithmic decision-making and a "right to explanation". In *ICML workshop on human interpretability in machine learning (WHI 2016), New York, NY. http://arxiv. org/abs/1606.08813 v1*, 2016.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[HAGM14] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Simultaneous detection and segmentation. In *European Conference on Computer Vision*, pages 297–312. Springer, 2014.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 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.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 21 May 2018].

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[KFF15] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[Khu15] Artem Khurshudov. Suddenly, a leopard print sofa appears. http://rocknrollnerd.github.io/ml/2015/05/27/leopard-sofa.html, 2015. [Accessed 10 January 2018].

[Kin] Pieter-Jan Kindermans. Learning how to explain neural networks: Patternnet and patternattribution.

[KS17] Jaedeok Kim and Jingoo Seo. Human understandable explanation extraction for black-box classification models based on matrix factorization. *arXiv preprint arXiv:1709.06201*, 2017.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[Le13] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[MDFF16] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[Meh15] Soroush Mehri. On binary classification with single-layer convolutional neural networks. *CoRR*, abs/1509.03891, 2015.

[MGB17] Konda Reddy Mopuri, Utsav Garg, and R Babu. Cnn fixations: An unraveling approach to visualize the discriminative image regions. 08 2017.

[mon] MongoDB for Giant Ideas | MongoDB. https://www.mongodb.com/. [Accessed 25 May 2018].

[MSM18] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1 – 15, 2018.

[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.

[MV16] Aravindh Mahendran and Andrea Vedaldi. Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision*, 120(3):233–255, 2016.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

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

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, Dec 2015.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM, 2016.

[RŠK08] Marko Robnik-Šikonja and Igor Kononenko. Explaining classifications for individual instances. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):589–600, 2008.

[SBM+17] Wojciech Samek, Alexander Binder, Grégoire Montavon, Sebastian Lapuschkin, and Klaus-Robert Müller. Evaluating the visualization of what a deep neural network has learned. *IEEE transactions on neural networks and learning systems*, 2017.

[SCD+16] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *See https://arxiv.org/abs/1610.02391 v3*, 2016.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[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.

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[SWM17] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*, 2017.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 04 2015.

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[Tur16] Ryan Turner. A model explanation system. In *Machine Learning for Signal Processing (MLSP), 2016 IEEE 26th International Workshop on*, pages 1–6. IEEE, 2016.

[VAMRL11] Alfredo Vellido Alcacena, José David Martín, Fabrice Rossi, and Paulo JG Lisboa. Seeing is believing: The importance of visualization in real-world machine learning applications. In *Proceedings: 19th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2011: Bruges, Belgium, April 27-28-29, 2011*, pages 219–226, 2011.

[Vin17] James Vincent. Facebook's facial recognition now looks for you in photos you're not tagged in, Dec 2017.

[VMGL12] Alfredo Vellido, José David Martín-Guerrero, and Paulo JG Lisboa. Making machine learning models interpretable. In *ESANN*, volume 12, pages 163–172, 2012.

[VTBE15] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

[Wel17] Adrian Weller. Challenges for transparency. *arXiv preprint arXiv:1708.01870*, 2017.

[YCN+15] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[ZCAW17] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.

[ZCW16] Luisa M Zintgraf, Taco S Cohen, and Max Welling. A new method to visualize deep neural networks. *arXiv preprint arXiv:1603.02518*, 2016.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[ZKL+16] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2921–2929, 2016.