Norwegian University of
Science and Technology

Building a platform for exploring and visualizing deep convolutional networks

Ole Øystein Barsch

Odin Eilertsen

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Deep artificial neural networks are showing a lot of promise when it comes to tasks involving images, such as object recognition and image classification. In recent years, there has been a steady increase in computing power, which has opened for the possibility of training deeper and more complex artificial neural networks. This, in addition to improved training methods, has been a major contributing factor for creating a new wave of AI within computer science. However, as artificial intelligence becomes more complex, it gets increasingly harder to explain an AIs reasoning. It is intriguing that computers achieve human-like results for image classification tasks, but from a research point of view, the reason why it performs so good might be even more interesting.

In this thesis, we aim to get a better understanding of deep convolutional neural networks. We attempt to increase our knowledge of these networks by creating a platform where one can apply different visualization methods on different pre-trained network architectures. First, we introduce the concept of convolutional neural networks, and how they work. We then give an introduction to the field of visualizing neural networks by explaining several state-of-the-art methods which aim to give a better understanding of neural networks. After introducing multiple methods, we explain our implementation of a selected few. We also give an introduction to the platform we created for handling the different visualization techniques. Included in this platform is a user interface, which simplifies the process of applying visualization techniques to different networks and retrieving the results. Finally, we examine the implemented techniques, while trying to explain their behavior and what information they can give us about a convolutional network. Additionally, we try to combine different visualization methods, to see if they offer any useful information beyond what each method offers individually.

Interpreting the results of visualizations proved to be a challenging task, but we still feel like there was some information to be gained from each distinct method. Certain techniques showed results which could be useful for troubleshooting faulty networks, while others indicated features which might be vital for correctly classifying images. Combining different techniques yielded results that were difficult to interpret clearly, but could prove to be a path worth researching further.

Sammendrag

Dype kunstige nevrale nettverk har begynt å vise stort potensiale når det kommer til oppgaver som omhandler bilder. Eksempler på slike oppgaver er objektgjenkenning, og klassifisering av bilder. I løpet av de siste årene har mengden med tilgjengelig databehandlingskraft økt, noe som har åpnet for muligheten til å trene dypere og mer komplekse kunstige nevrale nettverk. I tillegg har metodene for trening av slike nettverk blitt forbedret. Disse fremskrittene innenfor kunstig intelligens har vært store faktorer som har bidratt til å skape en ny bølge med kunstig intelligens innenfor datavitenskap. En ulempe relatert til mer komplekse modeller er at det blir vanskeligere å forstå en AIs virkemåte og resonering. Det er spennende å se at maskiner oppnår resultater på samme nivå som mennesker, men fra et vitenskapelig synspunkt er det enda mer interresant å forstå hvordan og hvorfor de oppnår så gode resultater.

I denne masteroppgaven ønsker vi å oppnå en bedre forståelse av konvolusjonære nevrale nettverks virkemåte. Vi forsøker å øke forståelsen av disse nettverkene ved å lage en platform hvor man kan bruke forskjellige visualieringsmetoder på forskjellige forhånds-trente nettverksarkitekturer. Først introduserer vi konseptet konvolusjonære nevrale nettverk og hvordan de fungerer. Videre vil vi gi en introduksjon til visualisering av nevrale nett, ved å forklare forskjellige metoder for visualisering. Disse metodene prøver å utdype forståelsen for hvordan nevrale nettverk fungerer. Etter dette vil vi gi en forklaring av hvordan vi har implementert noen av metodene. Vi vil også gi en introduksjon til platformen vi laget, som håndterer de forskjellige visualiseringsmetodene. Platformen inkluderer et brukergrensesnitt for å fremstille resultatene fra de forskjellige visualiseringsteknikkene. Til slutt vil vi undersøke resultatene til de forskjellige visualiseringsteknikkene for å se hva slags informasjon de kan gi oss om et nevralt nettverk. Vi vil også prøve å kombinere forskjellige visualiseringsteknikker for å se om det kan brukes til å hente ut informasjon som forklarer nettverkets oppførsel.

Å tolke resultatene fra de forskjellige visualiseringsteknikkene viste seg å være en vanskelig oppgave, men vi mener at det var mulig å hente ut relevant informasjon med hver enkelt metode vi implementerte. Enkelte av teknikkene ga resultater som kan være til bruk i forbindelse med feilsøking av nevrale nett, mens andre ga en indikasjon på hvilke trekk ved et objekt som var viktige for at nettverke skulle kunne klassifisere objektet. Kombinering av visualiseringsteknikker ga resultater som var vanskelige å tolke, men som også ga inntrykk av at det potensielt kunne vært spennende å utforske dette området videre.

Preface

This thesis was written as part of our master's degree in Computer Science at the Norwegian University of Science and Technology (NTNU) in Trondheim. It is our final project at NTNU, signaling the end of our five years of study. We would like to thank our supervisor Keith L. Downing for giving us the opportunity to work on this project, and for assisting us throughout our final semester at NTNU. We would also like to thank our family and friends for supporting us through our years of studies.

Contents

Abstract **i**

Sammendrag **iii**

Preface **v**

Table of Contents **viii**

List of Figures **xi**

List of Tables **xii**

List of Algorithms **xiii**

Glossary **xv**

1 Introduction **1**
- 1.1 Research Motivation 1
- 1.2 Research goal 3
- 1.3 Research questions 3
- 1.4 Outline 4

2 Background **5**
- 2.1 Artificial neural networks 5
  - 2.1.1 Artificial Neurons 6
  - 2.1.2 Activation functions 7
  - 2.1.3 Training neural networks 7
  - 2.1.4 The backpropagation algorithm 8
  - 2.1.5 Learning-rate and Optimization algorithms 9
  - 2.1.6 Regularization techniques 10
- 2.2 Convolutional networks 10
  - 2.2.1 Matrix operations 14
  - 2.2.2 Convolution activations 15
- 2.3 Network architectures 15
  - 2.3.1 VGG-16 16
  - 2.3.2 Inception V1 18
- 2.4 Technologies and frameworks 21
  - 2.4.1 Python 21

2.4.2 Javascript . . . 23
2.5 Related work . . . 25
2.5.1 Visualizing and Understanding Convolutional Networks . . . 25
2.5.2 Understanding Neural Networks Through Deep Visualization 27
2.5.3 Regularization techniques and image priors . . . 28
2.5.4 Visualizing and comparing convolutional neural networks . . 29
2.5.5 The LRP Toolbox for Artificial Neural Networks . . . 30
2.5.6 Explaining NonLinear Classification Decisions with Deep Taylor Decomposition . . . 31

3 Method **35**
3.1 Application design . . . 35
3.1.1 Backend . . . 36
3.1.2 Frontend . . . 36
3.2 Feature visualization . . . 36
3.2.1 The naive approach . . . 37
3.2.2 Transformations . . . 39
3.2.3 Utilizing TensorFlow for faster visualization . . . 41
3.2.4 Alternative parametrization spaces . . . 42
3.2.5 Decorrelating color channels . . . 45
3.2.6 Training the neural network . . . 47
3.2.7 Multiple optimization objectives . . . 47
3.2.8 Fine-tuning parameters . . . 48
3.2.9 Fetching image-examples of features . . . 48
3.3 Activation visualization . . . 49
3.4 Deep Taylor decomposition . . . 49
3.4.1 Generating a Deep Taylor graph . . . 50
3.4.2 Heatmap generation . . . 52
3.4.3 Challenges . . . 52
3.5 Combining visualization techniques . . . 53
3.6 Graphical user interface . . . 54
3.6.1 Prediction . . . 55
3.6.2 Activation visualization . . . 56
3.6.3 Feature visualization . . . 58
3.6.4 Deep Taylor Decomposition . . . 62
3.6.5 Challenges . . . 63
3.7 Loading pretrained networks . . . 63

4 Results and Discussion **65**
4.1 Feature visualization . . . 65
4.1.1 Features in Inception and VGG-16 . . . 66
4.1.2 More robust visualizations with stochastic transformations . 73
4.1.3 Advantages of alternate parameterization spaces . . . 75
4.1.4 Fetching examples from a dataset of images . . . 78
4.1.5 DeepDream . . . 79
4.1.6 Applications and usefulness . . . 79

4.2 Activation visualization . . . 81
4.3 Deep Taylor Decomposition . . . 83
4.3.1 Checkerboard artifacts . . . 85
4.3.2 Abnormalities in the Inception Network . . . 86
4.3.3 Recognizing general features using heatmaps . . . 87
4.3.4 Relevance through all layers . . . 88
4.3.5 Generating heatmaps for wrongly classified images . . . 90
4.3.6 Applications and usefulness . . . 92
4.4 Combining visualization techniques . . . 92
4.4.1 Visualizing individual filters . . . 92
4.4.2 Combining relevant filters . . . 93
4.4.3 Applications and usefulness . . . 95

5 Conclusion **97**
5.1 Future work . . . 99
5.1.1 Training convolutional neural networks . . . 99
5.1.2 Further improving the interpretability of feature visualization 99
5.1.3 Smart selection of neuron-groups for feature visualization . . 100

Bibliography **101**

A Activation Visualization **105**

B Visualization results **107**

C Screenshots of user interface **111**

List of Figures

2.1 Illustration of an artifical neuron. 6
2.2 Illustration of an artificial neural network. 6
2.3 Graph illustrating the ReLU activation function 7
2.4 Illustration of gradient descent converging towards the global minimum. 8
2.5 Illustration showing gradient descent with a high and low learning-rate. 9
2.6 Example of a convolutional layer, presented like an ordinary neural network. 11
2.7 Example of a simple convolution, with a 5x5x2 input, a 2x2x2 filter, a stride of [1,1] and no bias. 12
2.8 An example of how the input dimensions are retained when using zero-padding before a convolution 13
2.9 Example of pooling operations with a 2x2 pooling region 13
2.10 Figure showing an input, weight and output matrix 14
2.11 Example of input, weights and output when computing the gradients of a convolution. 15
2.12 Illustration showing the VGG-16 architecture 17
2.13 Illustration of an Inception module. 19
2.14 Model of the Inception V1 network. 20
2.15 Example of creating and executing a Tensorflow graph. Executing the code will result in the variable y being equal to 25. 22
2.16 The mathematical graph resulting from running the code in Figure 2.15. 22
2.17 Example of a react component. 24
2.18 Illustration of a deconvolutional network structure, inspired by the illustration in (Noh et al., 2015). 26
2.19 Example of an un-pooling performed on a max-pooling operation . . 27
2.20 Example of relevance propagation in an ordinary neural network. . . 31

3.1 Illustration of the dataflow within the application 36
3.2 Example of the jitter-transformation applied in the x-direction. . . . 39
3.3 The four interchangable sub-graphs, making up the Tensorflow graph structure used in feature inversion. 42
3.4 All the levels of a Laplacian pyramid with n = 3 are upsampled and combined into an optimized image 45
3.5 Figure displaying heatmaps when using max-pool and when using average pool. 51

3.6 The colormap used when creating heatmaps. 52
3.7 Illustration of the process used for calculating relevance scores 53
3.8 Screenshot of the initial page of the user interface 54
3.9 Screenshot of the component handling predictions. 55
3.10 Screenshot of the component for activation visualization. 56
3.11 Screenshot of the component which displays visualized activations. . 56
3.12 Screenshot of graphical representation of the selected network, which in this case is the Inception network. 57
3.13 Screenshot of an expanded Inception module in the visual representation of the network. 58
3.14 Screenshot of the component for setting parameters during feature inversion 59
3.15 Screenshot of the component for setting parameters when creating a "dreamed" image 61
3.16 Screenshot of the component for generating relevance heatmaps. . . 62
3.17 Screenshot of the component for displaying relevance heatmaps . . . 63

4.1 Visualization of Conv2d_2b_1x1 from the Inception network 67
4.2 Visualization of Conv1_2 from the VGG-16 network 67
4.3 Feature visualization of the first 5 channels from the earliest layers (1-10), throughout VGG-16 69
4.4 Feature visualization of the first 5 channels from layer 11-18 throughout VGG-16 70
4.5 Class-visualizations from two different CNNs 71
4.6 Traffic light class from VGG-16 visualized 72
4.7 Randomly selected features from the layer mixed_3b in the Inception network combined into new features. 73
4.8 Feature inversions from left to right: jitter = 0, jitter = 1, jitter = 10. 74
4.9 Feature inversions from left to right: angles = 0, angles = ($-5°$, ..., $5°$), angles = ($-180°$, ..., $180°$). 74
4.10 Feature inversions from left to right: scales = 1.0, scales = (0.8, ..., 1.2), scales = (0.1, ..., 1.9). 75
4.11 Feature inversions from left to right: padding = 0, padding = 12. . . 75
4.12 Feature visualization at various steps, utilizing different parameterization spaces. Learning-rates are set to: Standard = 0.2, Laplace = 0.05, Fourier = 3.0. The loss function is based on layer: mixed_4c, channel: 134, from the Inception network. 76
4.13 Feature inversion after 200 steps, using the same parameters as specified in Figure 4.12 77
4.14 Top 10 randomly extracted image-patches taken from the ILSVRC2017 test-set, that had the highest activation values for different channels in the Inception net. The mean output value from the channel is displayed under each image-patch. 78
4.15 Examples of the DeepDream algorithm run over an image of a flower bouquet, using various loss functions from both Inception and VGG-16 80
4.16 Example showing the top 10 activations from all layers 82

4.17 Example of heatmaps generated with Deep Taylor Decomposition. . 84
4.18 Figure displaying how checkerboard artifacts occur. 85
4.19 Comparison of heatmaps from both versions of Inception V1. 86
4.20 Images and resulting heatmaps. Each image is classified as an African grey parrot. 87
4.21 Heatmaps showing the relevance layer-by-layer 89
4.22 Examples of images which were wrongly classified, along with their corresponding heatmaps. 91
4.23 Graph showing the accumulated relevance for layer Conv5_2 in the VGG-16 network. 93
4.24 Example of visualizing for multiple filters 94

A.1 Example showing the top 10 activations from all layers in Inception V1 106

B.1 Visualizing the top 5 filters with the highest relevance scores for layers 1-6 in the VGG-16 network. 108
B.2 Visualizing the top 5 filters with the highest relevance scores for layers 7-12 in the VGG-16 network. 109
B.3 Visualizing the top 5 filters with the highest relevance scores for layers 12-18 in the VGG-16 network. 110

C.1 Screenshot of the prediction page 111
C.2 Screenshot of the feature visualization page 112
C.3 Screenshot of the DeepDream page 112
C.4 Screenshot of the page for visualizing activations 113
C.5 Screenshot showing the module for selecting layers within the network113
C.6 Screenshot displaying the page for Deep Taylor Decomposition . . . 114

List of Tables

2.1 A table displaying the properties of the Imagenet dataset. 16

3.1 All the different parameters that can be tuned in the GUI component created for Feature Visualization. 60
3.2 A table displaying the accuracies of the chosen networks. 64

4.1 List of classifications for the images in Figure 4.22 90

List of Algorithms

1 z^+-Rule . . . 33
2 $z^{\mathcal{B}}$-Rule . . . 33
3 Constraining activation values . . . 49
4 Relevance propagation in convolutional layers . . . 50
5 Relevance propagation in pooling layers . . . 51

Glossary

ANN Artificial neural network. 2, 5

API Application Programming Interface. 21, 23, 36

CNN Convolutional neural network. 1, 3, 5, 12, 13, 15, 18, 28–30, 46, 48, 65, 66, 68, 70–73, 79–81, 97, 98, 100

CSS Cascading Style Sheets. 23, 25

DFT Discrete Fourier Transformation. 43

DOM Document Object Model. 24

FFT Fast Fourier Transform. 43, 44

GAN Genarative Adverserial Network. 29

GPU Graphics Processing Unit. 10, 15, 21, 41, 66

HTML HyperText Markup Language. 23, 25

HTTP HyperText Transfer Protocol. 36

ILSVRC ImageNet Large Scale Visual Recognition Competition. 2, 15, 18, 90

JSON JavaScript Object Notation. 36

JSX JavaScript XML. 25

LRP Layer-wise Relevance Propagation. 30, 32, 97

ReLU Rectified Linear Unit. ix, 7, 12, 18, 19, 26, 32, 81

REST Representational state transfer. 36

RGB Red, Green, Blue. 16, 42, 43, 45

UI User Interface. 25

VGG Visual Geometry Group. xi, 2, 3, 16–18, 29, 30, 52, 53, 57, 63, 68, 72, 80–82, 90, 92, 93, 108–110

XML Extensible Markup Language. 25

Chapter 1

Introduction

This chapter presents a high-level introduction to the research field of convolutional neural networks and describes the motivation that lies behind this thesis. It also contains the primary research goal which serves as the cornerstone of our research. In order to achieve this goal, several research questions are introduced. By answering the research questions, we hope to come closer to the research goal and advance our understanding of convolutional neural networks.

1.1 Research Motivation

With recent technological improvements and new innovative ideas, deep artificial neural networks have begun to outperform humans in certain problem areas(He et al., 2015). This includes tasks that are easily solvable for humans but were previously thought to be computationally infeasible for machines. With a way to represent information inspired by research into the human brain, and a way to generalize based on a set of examples, deep neural networks are able to make sense out of complex, noisy and nonlinear data. These networks can be used to tackle a wide array of difficult problem areas, from recommendation systems to object detection and image classification.

A particular class of deep neural networks called convolutional neural networks (CNN) are especially good at analyzing visual imagery. What separates convolutional networks from other deep neural networks is the use of layers called convolutional layers throughout the network. These layers are taking into consideration the spatial information from the layer before so that they can represent different features of objects found in images. A convolutional neural network will try to learn a hierarchical representation of the input-images it has been trained on, where each layer in the network represents increasingly complex features. In the very first layers, these features are usually just simple concepts such as edges and

corners, while deeper down in the network, simple features are being combined in such ways that we are able to represent complex features like eyes and ears and so forth. Going even further, we can combine these into really high-level features such as faces or different types of animals.

In the last couple of years, we have seen convolutional neural networks achieve increasingly impressive results when it comes to object detection. The most recent winners of the ILSVRC competition(Russakovsky et al., 2015) have even been able to surpass human-level performance on several pattern recognition tasks. Seeing as these state-of-the-art models are able to classify images as well as humans, if not better, it has become an important question to discern how they are able to learn such intricate concepts. Due to its non-linear nature, it can be a hard problem to understand the correlation between input and output in an artificial neural network.

This problem has often been referred to as the black-box problem(Castelvecchi, 2016) of artificial neural networks. Although a deep neural network in theory is capable of approximating advanced non-linear functions, merely observing its structure and weights usually won't give much insight into how the function is being approximated. When the number of parameters is becoming really large, which is often the case with deep neural networks, it does not make things any easier. This is also the case with convolutional neural networks. Looking at the VGG-16 network(Simonyan and Zisserman, 2014) for example, a commonly used convolutional network for image recognition, it contains more than a 160 million different parameters. This is why we need smart techniques in order to properly visualize the inner workings of these networks, to gain further insight into how they operate.

Knowing how and why a convolutional network behaves the way it does and how decisions are being made is important for a multitude of reasons. Maybe the most obvious one is trying to understand how mistakes occur in order to prevent them. If an autonomous car, being driven by a deep neural network looking at visual input from a front-camera suddenly swerved into a tree at the side of the road, knowing why would obviously be of great interest. If we can grasp what parts of the network that constitutes different parts of the decision-making process, we might also get hints to whether or not the structure of the network could be optimized, by building upon it or getting rid of unnecessary parts. There could also be unforeseen insights to be gained by a deeper understanding of these networks. Seeing as artificial neural networks were inspired by biological processes in the brain, we might even be able to draw some parallels to the way the humans are processing visual information.

The quest for a better understanding of neural networks has yielded several visualization techniques(Alexander Mordvintsev, Christopher Olah and Mike Tyka, 2015; Erhan et al., 2009; Montavon et al., 2017; Olah et al., 2017; Yosinski et al., 2015; Zeiler and Fergus, 2014), which aim to visualize the internal workings of ANNs. Each method provides interesting visualizations for a given neural network,

displaying interaction between the learned weights in different ways. How useful would it be to have multiple visualization techniques easily available for trained CNNs, in order to see what the networks have learned, why images receive their classifications and what is really happening inside the networks?

1.2 Research goal

The main goal of this thesis is to be able to better understand a CNNs inner workings by creating a platform on which a user can explore a convolutional neural network using several different visualization techniques. Using said platform, the user should be able to better understand the relationship between an input image and predicted classification of said input. In addition, it should be possible to dive further into a network's architecture, and discover how the network represents the information it has collected through the training process. The visualization techniques we wish to implement are Feature visualization(Olah et al., 2017), DeepDream (Alexander Mordvintsev, Christopher Olah and Mike Tyka, 2015), Activation Visualization(Yosinski et al., 2015) and Deep Taylor Decomposition(Montavon et al., 2017). We want to apply these methods to different CNN architectures, in order to see what kind of information that can be gained from visualizing convolutional neural networks.

1.3 Research questions

In order to concretize the objectives for this project, we defined a set of research questions which we hope to answer with our research, and to help maintain continuity throughout our project without derailing towards unrelated tasks and subjects.

Can we create a platform where users can apply different visualization techniques on networks with different architectures? In order to evaluate different visualization techniques, we want to create a system with a user interface, where one can try out and switch between different visualization methods, and compare their results. The system should support multiple convolutional networks with different architectures out-of-the-box, implying that the user has to do no additional modifications when switching between different architectures. For this thesis we limit ourselves to two different network architectures; Inception V1(Szegedy, Liu, et al., 2015) and VGG-16(Simonyan and Zisserman, 2014).

Does utilizing visualization techniques yield any information which can be used to explain a neural networks behavior, and if so what does it tell us? There exist several methods for visualizing neural networks, but an important

question remains as to how useful the retrieved information is, and what it implies. We want to answer the question if the results from visualizing neural networks can be used for practical purposes, such as troubleshooting a faulty neural network or improving an existing architecture.

Is there any additional information to be gained by combining different visualization techniques? Each visualization technique gives its own piece of information about a network. We wish to see if there is any additional information to be gained by utilizing a combination of different visualization techniques. In particular, we want to see if Deep Taylor Decomposition can be used to guide feature visualizations in order to make the results easier to interpret.

1.4 Outline

This section gives a short introduction to each of the following chapters and describes the overall structure of the thesis.

Background contains all information which we deemed necessary for understanding the field of visualizing neural networks. This chapter includes an introduction to basic concepts in neural networks, as well as related work in the field.

Method describes the implementation details for the platform we created. Here we explain how the system is structured, and how different visualization methods were implemented.

Results and Discussion holds the results of utilizing the different methods we implemented, as well as our evaluation of said results. Each method and its contributions towards our research questions are evaluated on their own before we look at the results of combining different techniques.

Conclusion wraps up our research in relation to our research goal and research questions. We review what we have learned from our research, and to which degree we were able to contribute to the field of visualizing neural networks. We present a few directions for future work, which look promising and intriguing.

Chapter 2

Background

This chapter contains an introduction to a few central concepts, network architectures, and technologies that would be useful to understand, as the rest of the project is built upon these. For this thesis, we assume the reader has prior knowledge when it comes to artificial neural networks and their capabilities. We give a short introduction to ANNs, in order to explain the terminology used in later sections and to refresh the readers understanding of neural networks. Following the explanation of neural networks, we explain one special category of ANNs, known as convolutional neural networks. This class of neural networks is used in particular for tasks involving image data. After giving an introduction to convolutional networks, two different network architectures are explained in detail. These two network structures are used as examples when exploring different visualization techniques and are therefore explained thoroughly. Also introduced in this chapter are a few technologies accompanied by a short explanation of why we selected these technologies instead of other alternatives.

The related works section explores the research area of visualizing CNNs by presenting several papers which describe different methods for achieving a better understanding of how convolutional networks work. The section aims to give a perspective on the current state-of-the-art visualization techniques and introduce techniques which are implemented in later sections of this thesis.

2.1 Artificial neural networks

This section gives a short introduction to what artificial neural networks are, how they work and how they are trained. A basic understanding of neural networks is crucial for understanding the work done in this thesis.

Artificial neural networks are a type of system used in machine learning, which is

inspired by the human brain. When we mention neural networks in this thesis, we refer to artificial neural networks and not their biological counterparts, unless specified. Neural networks in biology consist of multiple neurons. A neuron has one or several parent neurons to which it is connected, as well as one or several children. When a neuron receives a signal from its parents, it can choose to propagate the signal to its children. Artificial neurons aim to mimic this behavior.

2.1.1 Artificial Neurons

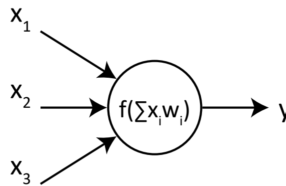


Figure 2.1: Illustration of an artifical neuron.

An artificial neuron has several input connections, an activation function, and one output connection. Each input connection x_i has a corresponding weight w_i. In order to determine if the neuron should fire, it performs the calculation shown inside the neuron in Figure 2.1. The weighted input values are summed together, before being passed through a function $f(x)$, also known as an activation function. The activation function determines the value which the neuron will output.

An artificial neural network is utilizing several neurons in groups known as layers. Usually, a neural network consists of one input layer, several "hidden" layers and one output layer.

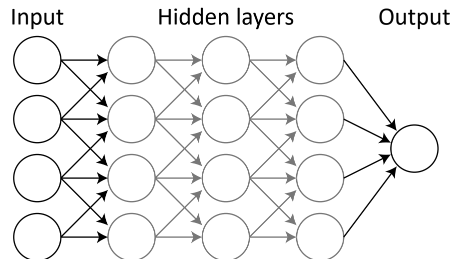


Figure 2.2: Illustration of an artificial neural network.

If all neurons in one layer share their outputs with all neurons in the next layer, we call it a fully connected layer. By utilizing a structure of neurons, we can create a system which can represent advanced non-linear functions or tasks. In Figure 2.2

there is only one neuron in the final layer. In networks performing classification tasks, there usually exists one neuron for each distinct class.

2.1.2 Activation functions

The most primitive activation function is the linear activation, where $f(x) = x$. The output of a linearly activated neuron will always be equal to the sum of all weighted input values. However, using a linear activation function makes it hard for the model to generalize for non-linear data. This is where non-linear activation functions come in.

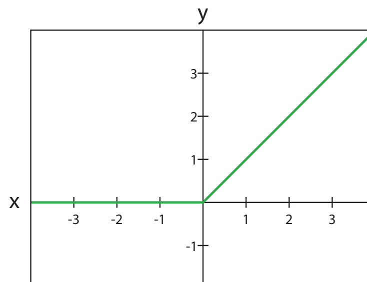


Figure 2.3: A graph illustrating the ReLU activation function. x represents the input, and y the output. The green line represents the ReLU function.

For this thesis the only relevant activation functions are the ReLU(Glorot et al., 2011) function which is defined as $f(x_i) = max(0, x_i)$, and the softmax function, defined in Equation 2.1. ReLU is used as an activation function for hidden neurons, while softmax is used at the output layer in classification tasks.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.1}$$

2.1.3 Training neural networks

In order for a neural network to be able to perform a certain task, one needs to "train" the neural network. This is done by adjusting the weights of the network until the network is sufficiently "good" at its task. The training process of standard neural networks is performed through what we refer to as supervised learning. We have a set of training examples, consisting of a set of input values, and the corresponding values that we want the network to output. The first thing that happens is called a forward pass, which is to send the input values from one training sample through the network and then calculate the output value. The correct output from the example, along with the actual output from the network are used

to calculate the training error also known as loss, by using a loss-function. The goal of the training process is to minimize this function. In order to adjust the weights in a way which minimizes the error, we use the backpropagation algorithm. To measure how well a network does, one splits the training data into a training set and a test set. After each training epoch, the test set is evaluated using the trained model. The accuracy one achieves when evaluating the test set shows how well the trained model generalizes for previously unseen data. If the accuracy of the test set starts to decrease compared to previous iterations, the system could be in the process of over-fitting. Over-fitting occurs when a model is too complex and is formed to only predict the training data really well, rather than generalize for all possible data.

2.1.4 The backpropagation algorithm

In order to minimize the loss-function, gradient-descent is used. The global minimum of the loss-function is not known, but we can approximate it using the gradient of the loss function.

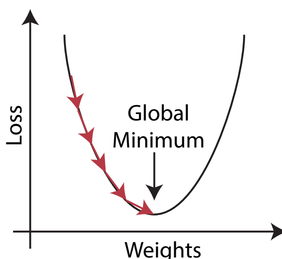


Figure 2.4: Illustration of gradient descent converging towards the global minimum.

First, we initialize the network with random values for all weights. This is what we refer to as our starting point. Gradient descent calculates the gradient of the loss function with respect to the weights. Since we have several different weights in the network, the gradient is a set of partial derivatives with respect to the weights. The algorithm then adds a fraction of the gradient to the initial starting point. The size of the fraction is what we refer to as learning rate. This will reduce the value of the loss function. By repeating this process we move closer towards the global minimum of the loss function.

Backpropagation is a method which can be used to calculate the gradients in a neural network efficiently. When training neural networks, backpropagation is used along with gradient descent. Calculating all partial derivatives at once is a complex task. Instead, backpropagation propagates the value of the loss-function backwards layer by layer and calculates the derivative of the loss with respect to the weights at each layer.

2.1.5 Learning-rate and Optimization algorithms

The amount we choose to alter the weights in each iteration has a huge impact on how fast we reach the global minimum of the loss function. A system with a high learning-rate might never reach the global minimum, while a system with a low learning-rate can get ”stuck” in a local minimum, as seen in Figure 2.5.

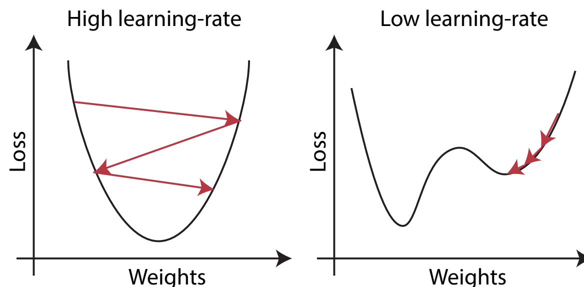


Figure 2.5: Illustration showing gradient descent with a high and low learning-rate.

In order to combat these issues, there exist several methods which utilize dynamic learning-rates. These methods are also referred to as optimization algorithms or optimizers.

Adam

Adam(Kingma and Ba, 2014) is an example of such a method, which uses a form of momentum to guide the learning process, with adaptive learning-rates for each individual weight. It is shown to work well in practice, outperforming other learning algorithms using adaptive learning-rates. The update rule in the Adam algorithm is computed with the equation below, where θ represents the weights, η the learning rate and ϵ a small number, typically 10^{-8}:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.2}$$

The change in each individual parameter is affected by both the decaying averages of past gradients, represented by $\hat{m}_t$ and the decaying averages of past gradients squared, represented by $\hat{v}_t$. These values are computed with the following equations, where g_t is the gradient we get from the backpropagation algorithm at step t:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t \tag{2.3}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t}, \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{2.4}$$

We often refer to m_t as the first moment (mean), and v_t as the second moment (variance). The two beta parameters are used as decay-rates for each of these terms, with the recommended default values set to be $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

2.1.6 Regularization techniques

As the complexity of networks increases, so does the possibility for over-fitting. Regularization techniques are methods which aid in the training process of neural networks in order to help the model generalize better.

L2 regularization

L2 regularization(Ng, 2004) is a regularization technique which is often used to combat over-fitting. The idea is to add an additional term to the loss-function of a network which combats high magnitude weights, something that often occurs in over-fitted networks.

$$Loss = L + \lambda \sum_i w_i^2 \tag{2.5}$$

Equation 2.5 shows how the error is calculated. L is equal to the error before applying L2 regularization. $\sum_i$ iterates over all weights in the entire network. Only a fraction of the sum of squared weights is added to the loss, and λ is used to define the size of this fraction. If the total weights in the network increase in magnitude, so does the total loss of the model.

2.2 Convolutional networks

Recent increases in computational power, and improvements on training neural networks, such as using the GPU for training(Steinkraus et al., 2005) have resulted in deeper and more advanced models being used within machine learning. Convolutional networks(LeCun et al., 1998) are a subset of neural networks, in which at least one layer of the network performs convolutional computations of its input, which are then passed onto the next layer. A convolution in this context is the process of applying a filter to the input tensor. The filter corresponds to what in a conventional neural net is referred to as weights. A filter can be seen as a three-dimensional tensor. Its size may vary, but its dimensions are smaller than those of the input, and usually the width and height are the same. The filter is used by applying it to one section of the input, calculating the value by multiplying the

input within the section with the filter and moving the filter to its next position before calculating the next value. This process goes on until the filter has reached the end of the input. The pacing of which the filter is moved is called the stride. A stride of [2,2] would suggest that the filter moves two steps at a time in the X direction of the input until it reaches the end of dimension X. Then it moves two steps in the Y direction before it starts moving along the X-axis once more.

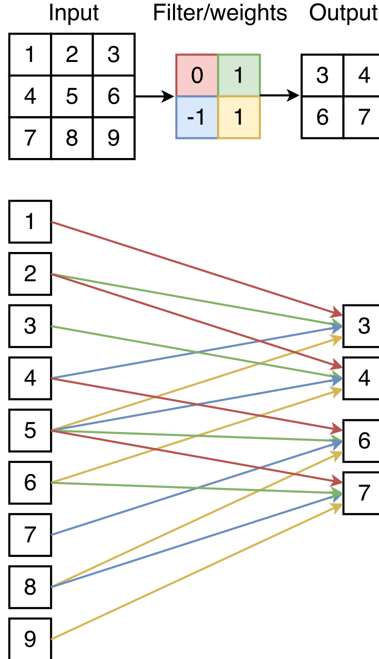


Figure 2.6: Example of a convolutional layer, presented like an ordinary neural network.

Convolutional neural networks exploit two principles; sparse connections and parameter sharing. Sparse connections reduce the number of weights the network has to learn. Parameter sharing further decreases the number of parameters and gives the ability to recognize patterns in multiple locations. Figure 2.6 visualizes these two principles. The network illustrated uses a stride of [1,1]. Each output node has only a number of input values equal to the total filter size, as opposed to using all inputs which would make it a fully connected layer. Each weight has a unique color, which is used to show how the parameters are shared between different nodes.

Lets say we have an input I, a filter F with dimensions [n,m,d] and an output O. The value of $O_{i,j,l}$ would then be

$$O_{i,j,l} = (\sum_{z=0}^{d-1} \sum_{y=0}^{m-1} \sum_{x=0}^{n-1} I_{x+a,y+b,z} * F_{x,y,z,l}) + B_l \tag{2.6}$$

Where l is the filter index, i and j represents the coordinates of the output, B is the bias for the filter and [a,b] are the offsets generated by moving the filter according to the stride. With a stride of [e,f] the values of a and b would be

$$a = e * i \tag{2.7}$$

$$b = f * j \tag{2.8}$$

The calculations below show how the value in Figure 2.7 is calculated.

$$O_{0,0,0} = \sum_{z=0}^{1} \sum_{y=0}^{1} \sum_{x=0}^{1} I_{x,y,z} * F_{x,y,z,0} \tag{2.9}$$

$$O_{0,0,0} = (1*2+2*0+(-5)*2+4*0)+(2*1+0*(-1)+3*1+6*1) = 3 \tag{2.10}$$

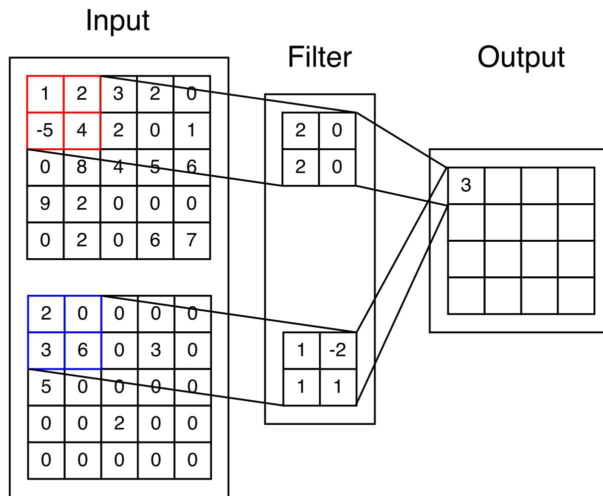


Figure 2.7: Example of a simple convolution, with a 5x5x2 input, a 2x2x2 filter, a stride of [1,1] and no bias.

The result of the convolution is a 4x4x1 tensor. An ordinary convolution operation reduces the dimension of the input, in this example from 5x5 to 4x4. As this is not always desired, one can apply a method called padding on the input to achieve an output with the same dimensions as the input. An example of this is shown in Figure 2.8.

Convolutional layers are usually followed by a ReLU activation function. In addition, there is also another type of layer found in CNN's known as a pooling layer. This layer reduces the dimensions of the input by pooling a region of the input.

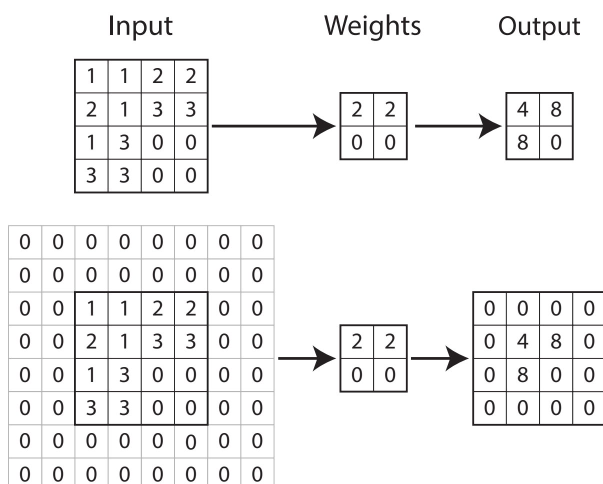


Figure 2.8: An example of how the input dimensions are retained when using zero-padding before a convolution. The top figure shows a convolution without padding, while the bottom figure uses zero-padding. This particular convolution uses a stride of 2.

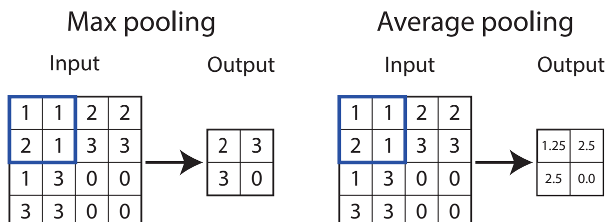


Figure 2.9: Example of pooling operations with a 2x2 pooling region illustrated as the blue boxes. The top figure shows a max-pooling operation, while the bottom figure displays a average pooling operation.

The most used type of pooling layer is known as max-pooling, in which the layer selects the highest value within the pooling region, which is then passed onto the output. Another type of pooling layer is average pooling. This operation returns the average value of the pooling region. Average pooling is often used after the last convolutional layer, in order to transition from convolutional layers to fully connected layers. Examples of pooling operations are displayed in Figure 2.9.

A CNN can be seen as two operations. The first operation is feature extraction, using convolutions. The second operation is classification, where the learned features are mapped to the output of the network. In a regular convolutional network, this is done by "flattening" the input to a 1x1xN tensor and then applying one or several fully-connected layers, often followed by a layer utilizing the softmax-activation

function.

2.2.1 Matrix operations

In most neural network libraries, convolutions are done using matrix operations. As these operations are used in Section 3.4 it would be an advantage to understand the basic concept. Figure 2.10 shows an example of the input, weights and output of a convolution.

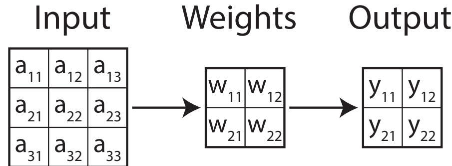


Figure 2.10: Figure showing an input, weight and output matrix. This example uses a stride of 1.

In order to utilize matrix operations, we first need to write the input matrix as a vector I. Then we create a sparse matrix from the initial set of filter weights which we call V. Calculating VI will give us the convolution of said input and weights. This process can be seen in the example Equation 2.11. Each element in the final vector corresponds to Equation 2.9. The final vector can be reshaped into a 2x2 matrix.

$$\begin{aligned}
&\begin{bmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ a_{33} \end{bmatrix} \\
&= \begin{bmatrix} a_{11} * w_{11} + a_{12} * w_{12} + a_{21} * w_{21} + a_{22} * w_{22} \\ a_{12} * w_{11} + a_{13} * w_{12} + a_{22} * w_{21} + a_{23} * w_{22} \\ a_{21} * w_{11} + a_{22} * w_{12} + a_{31} * w_{21} + a_{32} * w_{22} \\ a_{22} * w_{11} + a_{23} * w_{12} + a_{32} * w_{21} + a_{33} * w_{22} \end{bmatrix} = \begin{bmatrix} y_{11} \\ y_{12} \\ y_{21} \\ y_{22} \end{bmatrix}
\end{aligned} \tag{2.11}$$

Computing the gradient of a convolution is done in similar fashion. It is still a convolutional operation, but the weights and input change.

The input is a zero-padded version of the output from the original convolution, and the weights have been rotated 180 degrees.

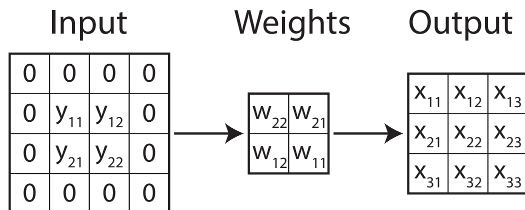


Figure 2.11: Example of input, weights and output when computing the gradients of a convolution.

2.2.2 Convolution activations

The output of a neural network layer is often referred to as its activation. The activations of a convolutional layer differ from those of a conventional fully connected layer in a special way, which one can exploit. In a fully connected layer, the order of the activations is irrelevant. In a convolutional layer, Each filter creates one two dimensional output, as shown in Figure 2.7. These activations retain spatial information and can therefore be displayed graphically. This representation is used in (Stutz, 2014) to get a better understanding of what happens inside of a convolutional network. The details of how to visualize the activations are described in Section 3.3

2.3 Network architectures

This section aims to give an understanding of two particular convolutional network architectures. These two architectures are both state-of-the-art CNNs with high accuracy when it comes to classifying images. A drawback of using state-of-the-art architectures is the computing power needed for the training process. The creators of the Inception network (described in Subsection 2.3.2) spent two weeks training their network, utilizing multiple high-class GPUs. Luckily there exist pretrained versions of both networks, which are compatible with our choice of deep learning framework, which is discussed more in Subsection 2.4.1.

Before we explain the two architectures, we give a short introduction to the Imagenet dataset(J. Deng et al., 2009). Imagenet is a database of images, order by different classes. The Imagenet dataset is a subset of the image database, used in the ILSVRC(Russakovsky et al., 2015), which is a competition for creating the best large scale system for image recognition. Through the last years of the ILSVRC, the dataset has remained unchanged. Table 2.1 shows the number of images for the different subsets in the Imagenet dataset.

There are a total of 1000 different categories of images. The validation and test set are hand labeled, thus ensuring the correctness of the label assigned to each

total number of classes/categories	1000
Training set size	1.2 million images
test set size	150 000 images
validation set size	50 000 images

Table 2.1: A table displaying the properties of the Imagenet dataset.

image. The balance of the dataset is not stated, meaning the dataset could be unbalanced by having more images for certain categories. All external images used in this thesis are extracted from the Imagenet dataset. In order to use the images in our research, we had to comply with the Imagenet terms of access [1].

2.3.1 VGG-16

The VGG-16 architecture is a result of the paper Very deep Convolutional Networks for large-scale image recognition(Simonyan and Zisserman, 2014) in which the authors aim to evaluate the benefit of increasing the depth of a convolutional neural network. The paper implements several networks with a different number of weighted layers. VGG-16, as the name suggests has 16 weighted layers. The structure of the network is shown in Figure 2.12. Images fed into the network are preprocessed by subtracting the mean RGB value of the training set. Convolutional layers are using a padding size which keeps the output dimensions equal to the input dimensions. Max-pooling is performed using a 2x2 window with a stride of 2. All convolutions are performed using a filter size of 3x3 with a stride of 1.

[1] `http://www.image-net.org/download-images#term`

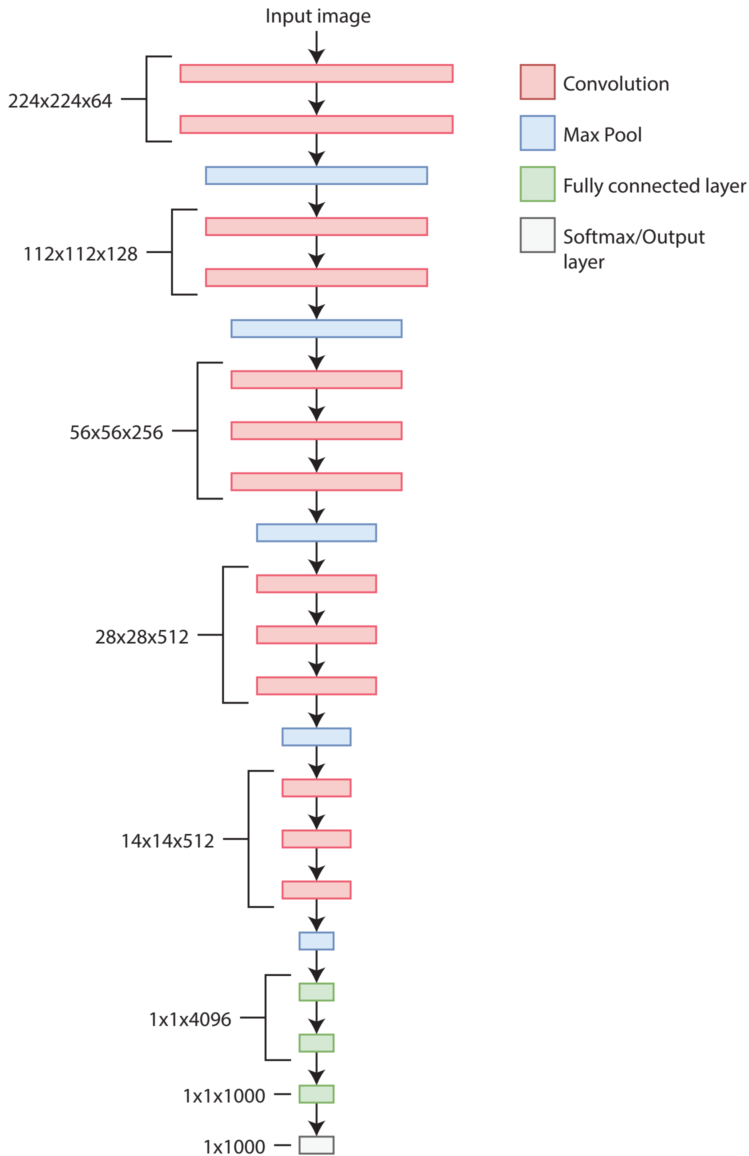


Figure 2.12: Illustration showing the VGG-16 architecture. The numbers show the size of each layer. The first two numbers indicate the width and height when training. The third number indicates the number of filters used in each layer.

Earlier CNNs used larger receptive fields than the VGG network. In this context, the receptive field is the dimensions of the weights used in a convolution. The convolutions in the VGG network all use filters with dimensions 3x3, which means they have a 3x3 receptive field. The paper from which the architecture originates argues that by stacking convolutional layers with small receptive fields they achieve the same result as a higher level receptive field. Stacking several smaller receptive fields has several advantages over using one large receptive field. Each convolution is followed by a ReLU function, which makes the decision function more discriminative. Having more layers will increase the use of the ReLU function, thereby making the decision function even more discriminative. The number of parameters needed is also reduced, which is another advantage.

2.3.2 Inception V1

Going deeper with convolutions(Szegedy, Liu, et al., 2015) describes an image classification system created and used by Google, that was also used in their open source release of DeepDream(Alexander Mordvintsev, Christopher Olah and Mike Tyka, 2015). The system, which has received the code name "Inception" was used in the ILSVRC 2014 Classification Challenge(Russakovsky et al., 2015). One issue with convolutional neural networks is attempting to use all existing computation power available efficiently. The most effective methods use dense matrix multiplication, while CNN's often are sparsely connected, and therefore are represented by sparse matrices. Google's team solved this issue by clustering sparse matrices into smaller more dense sub-matrices which are then used for computations.

The structure of the network can be seen as a composition of multiple modules. The main module of the network is referred to as an "Inception-module." This module consists of multiple operations which are run in parallel. At the end of the module, the results of all parallel operations are concatenated and sent to the next layer of the network. The operations that run in parallel are 1x1, 3x3 and 5x5 convolutions, in addition to a max-pooling operation. In the earlier layers of the network, correlated activations/pixel values can often be found close together in concentrated regions. These units will be picked up by the 1x1 convolutions. In addition, there are some correlations which are not as concentrated, to which the 3x3 and 5x5 convolutions will respond. The max-pooling is justified by arguing that it works well in similar networks, and therefore also should work well in this architecture.

The output-dimensions of an inception-module increase rapidly when chaining together several modules. This results in a lot of computations for the next layer, which then will increase the depth even further. To combat the increasing computational amount, a 1x1 convolution is run before the 3x3, 5x5 and max-pooling operations. The 1x1 convolution, which is sometimes referred to as a Network in Network(Lin et al., 2013) layer, is used to reduce the dimensions of inputs for computationally expensive operations. In order to better understand this concept,

we can look at a practical example in the Inception V1 network. The first Inception module has an output dimension of 28x28x256. Feeding this output straight into the 3x3 and 5x5 convolutions in the next Inception module would be computationally expensive. Instead one can apply a 1x1 convolution with 128 filters, thereby reducing the dimensions to 28x28x128. Finally, after each convolutional operation, the system uses a ReLU-activation function in order to introduce non-linearity.

Figure 2.13 shows a visual representation of the layers within an inception module. The red nodes perform a convolutional operation followed by a ReLU activation, while the blue node performs max-pooling. The yellow nodes denote the input and output of the module.

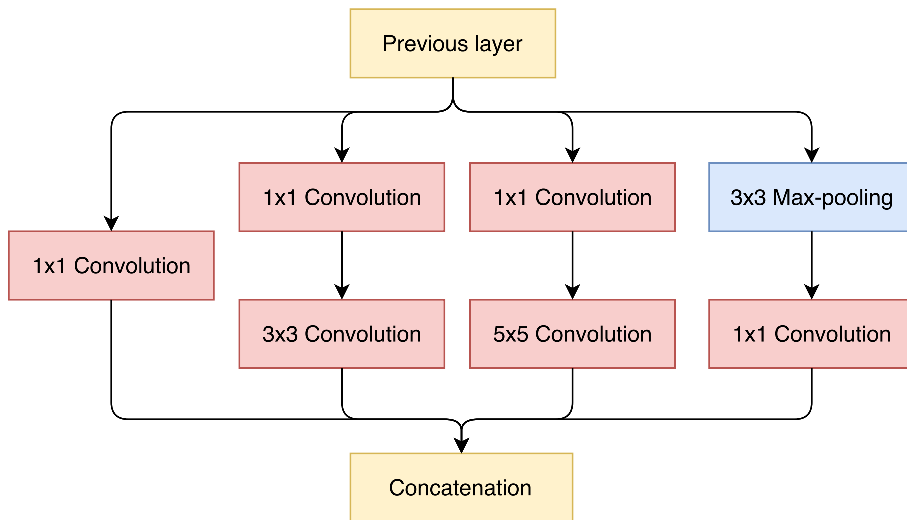


Figure 2.13: Illustration of an Inception module.

In addition to the Inception-module, the network also consists of a few ordinary conv-net operations in the early layers. This is due to memory restrictions in the training process, as the team realized it would be expensive to implement the Inception module in the early layers. Another enhancement which is implemented in the system are two auxiliary classifiers. These are connected to modules in the middle of the network, and predict the output based on its input, much like the final prediction layer. The loss of these outputs is then added to the total loss of the system. This encourages discrimination in the earlier layers of the system, as well as adding regularization and increasing the gradients during backpropagation.

The team behind the Inception architecture has created several iterations of the Inception-net(Szegedy, Vanhoucke, et al., 2015). Later iterations involve more complex inception modules with a higher number of filters and convolutional layers. The networks with their trained weights are available for use to the public. For this thesis, the first iteration of the Inception network was used. The architecture

is sufficiently complex, meaning the network is able to learn different features and concepts from the training dataset. At the same time is the architecture still relatively simple to explain, compared to later iterations.

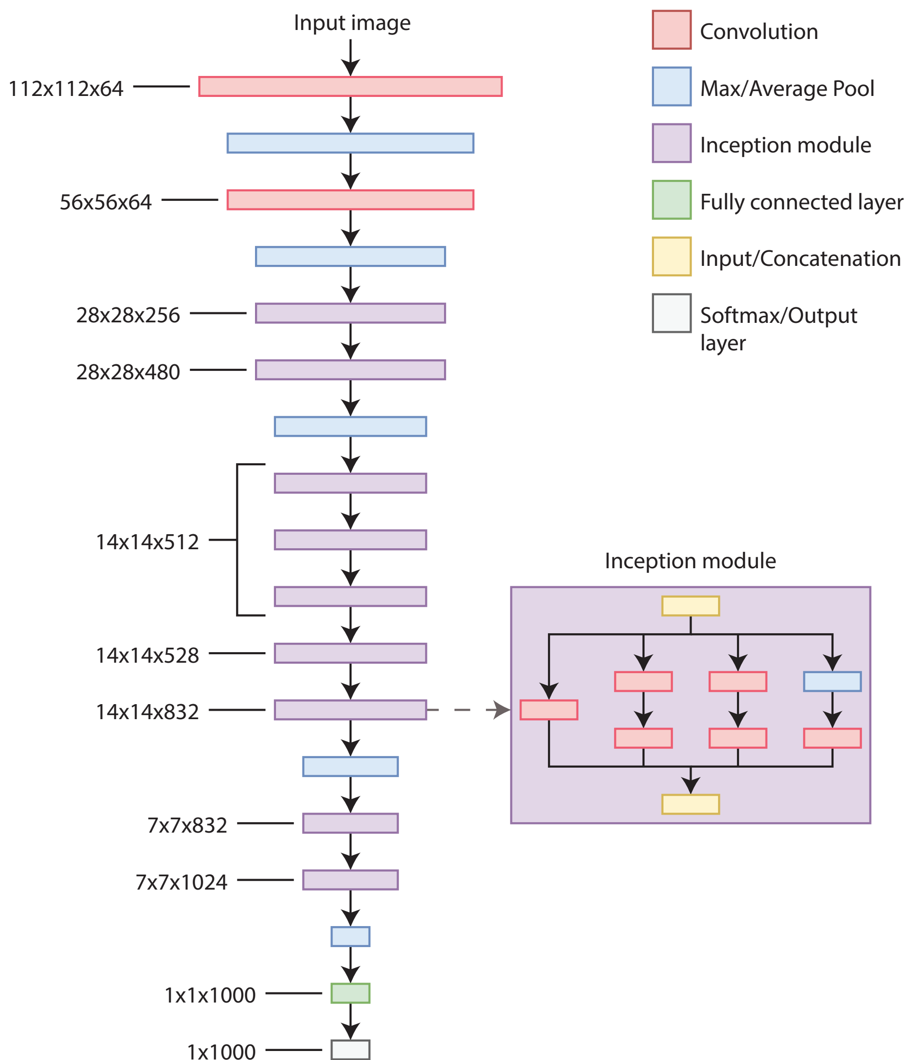


Figure 2.14: Model of the Inception V1 network.

Figure 2.14 shows a model of the first Inception network. The first convolutional layer uses a 7x7 filter with a stride of 2, and no padding. This convolution reduces the input size from the original 224x224 to 112x112. The next convolutional layers use 3x3 filters with a stride of 1x1 and padding which retains the original dimensions. All pooling layers apply max-pool with a size of 3x3 and a stride of 2, except

for the last layer, which applies average pooling on a size of 7x7 and a stride of 1. The auxiliary classifiers from the original model are excluded from this illustration, as they are only relevant during the training process.

2.4 Technologies and frameworks

This section gives a short introduction to the different programming languages and frameworks used in this thesis. In addition, we give an explanation for choosing each technology.

2.4.1 Python

Python is a high level, object-oriented programming language. It was used for this project as it supports a wide range of machine learning frameworks. A High-level language refers to a language with a high level of abstraction from machine language. This simplifies the process of working with neural networks, as one can write code on a model level, rather than a machine level. The best alternative to Python would have been C++, but neither of the authors had previous experience using it. In addition, it is much closer to machine language, which was another argument favoring Python. Java was also considered as an alternative, but the support for machine learning frameworks is lacking, and the structure of Java projects is more complex compared to Python, which seemed like an unnecessary burden given the relatively small scale of our project.

TensorFlow

Tensorflow(Martín Abadi et al., 2015) is an open-source library for numerical computations using data flow graphs. The process of creating and training neural networks is made a lot simpler with the built-in methods of Tensorflow. The system was created by the Google Brain team, whose goal is to improve machine intelligence. Tensorflow comes with several APIs in different programming languages such as Python, C++, Java and Go. For this project, the Python API was used, as it is the official release, and the team felt comfortable using it as a result of previous experiences. The choice of API does not influence the run-time of the code since all APIs compile their code to C++ at the lower levels. Tensorflow allows for running computations using one or several GPUs, which reduces the time spent on executing computations, and leaves more time for research and testing.

Tensorflow operates by chaining together operations on tensors, creating a computational graph which can then be executed using what Tensorflow refers to as a *session*. Figure 2.15 shows an example of Tensorflow code, which generates and executes a Tensorflow graph.

```
import tensorflow as tf

A = tf.placeholder(tf.int16)
B = tf.placeholder(tf.int16)
C = tf.constant(5, tf.int16)
D = tf.add(A, B)
E = tf.multiply(C, D)
with tf.Session() as sess:
    Y = sess.run(E, feed_dict={A: 2, B: 3})
```

Figure 2.15: Example of creating and executing a Tensorflow graph. Executing the code will result in the variable y being equal to 25.

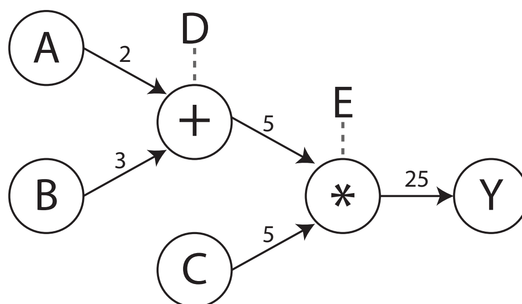


Figure 2.16: The mathematical graph resulting from running the code in Figure 2.15.

First three tensors, A, B, and C are defined. A and B are added together, before being multiplied with C. Tensorflow has three general types of tensors; placeholders, constants, and variables. When initializing a tensor, one needs to define the variable type of the tensor, such as integer, float and so forth. If the tensor has a multidimensional shape, the shape has to be passed into the initializing function as well. Placeholder tensors, as the name suggests, need to be fed a value when executing the computational graph. The values for placeholder tensors are submitted using a parameter called *feed_dict* as shown in Figure 2.15. Constants are defined once, and therefore need their value defined in the initializing function. Variable tensors hold a state, which is changeable within an initialized session. It requires an initial value in addition to a shape and type. An example of the usage of the variable tensor are the weights within a neural network. The strides and padding of a convolution would be constant tensors, while the input image would be a placeholder.

When training a neural network in Tensorflow, all weights are stored as Tensorflow variables. At the end of the training session, the values of all weights can be stored in a checkpoints file. The trained network can then be restored at a later point.

There are several alternatives to Tensorflow, such as Torch[2], PyTorch [3], Keras[4], Theano[5] and Caffe[6]. All these frameworks deliver much of the same features and methods. The decision to use Tensorflow was based on several criteria. First, Tensorflow is a low-level implementation, when compared to other systems such as Keras. This allows for more flexibility and space to create a tailored solution for the project. The team thought the option of spending some time to learn Tensorflow was better than using a higher level system, which in later stages could prove to lack features needed for the project. Additionally, the team had previous experience with using Tensorflow for generating neural networks.

A byproduct of Tensorflow being created by employees at Google was extensive documentation of its APIs, as well as tutorials and guides from members of the machine learning community. This was important as it would make it easier to fully understand the framework, implementing desired features and solving common problems which others had encountered before.

Flask

Flask[7] is a micro web framework for Python. A micro-framework is a framework which is not dependent on any distinct libraries. Flask delivers basic functionality needed to create a simple web server. It offers support for extensions which can add new functionality to the application, or extend existing functions. As the selected Tensorflow API uses Python, it was only natural to select a web-framework which did the same, in order to simplify the communication between the Tensorflow model and the server.

2.4.2 Javascript

Javascript, CSS and HTML are the core technologies for creating web-pages. In this thesis several Javascript libraries are used which combine HTML, CSS and Javascript.

Node

Node, also known as Node.js is an open-source Javascript run-time environment. It can be run on all common platforms, including Windows and Linux. For this thesis, Node is used for development of the user interface. Node comes with a package manager, which makes it easy to acquire and manage different packages

[2] http://www.torch.ch/
[3] https://pytorch.org/
[4] http://keras.io
[5] http://deeplearning.net/software/theano/
[6] http://caffe.berkeleyvision.org/
[7] http://flask.pocoo.org/

needed for development. Node is also used to host the user interface, in order to make it accessible to the end user.

React

React[8] is a library written in JavaScript, which is used for creating user interfaces. The library is written and maintained by Facebook. It is widely used in different projects, ranging from small hobby projects to large websites such as Instagram[9]. As a result of being developed by Facebook, React has a huge following, which in turn created a big community with guides and libraries for use with React. React is flexible as it can be used to create a standalone user interface only written in react, or be used in combination with other libraries.

A standalone React application is also called a single-page application. React works by manipulating the Document Object Model, or DOM for short. This is done by maintaining a virtual DOM in addition to the actual DOM, and only changing the real DOM when changes occur in the virtual DOM, as opposed to recreating the entire DOM each time a change occurs.

```
import React, { Component } from "react";

class HelloWorld extends Component {
    state = {
        text: "",
    }

    showText() {
        this.setState({
            text: "Hello␣world",
        })
    }

    render(){
        return(
            <div>
                <p>{this.state.text}</p>
                <button
                    label = "button"
                    onClick = { this.showText }
                />
            </div>
        )
    }
}
```

Figure 2.17: Example of a react component.

[8] https://reactjs.org/
[9] https://www.instagram.com/

Figure 2.17 Shows an example of a react component written in Javascript. This component displays a button. Upon pressing the button, a text will appear with the value "Hello world." The component consists of an internal state, a function showImage, which changes the internal state, and a render function, which returns JSX code. JSX is an abbreviation for Javascript XML, and is an extension of HTML with Javascript. Upon clicking the button, the function showText is invoked, which changes the internal state. Each time the state of a component changes, the render function is called. When comparing React to the Model-view-controller architecture(Krasner, Pope, et al., 1988), a component entails both the view as well as the controller, thereby simplifying the data flow within the application. For this project we had no need for an advanced user interface, we only needed to be able to display the results from our model, as well as to hold a basic state within the application. This constraint made React the best choice of library, compared to other alternatives such as Django[10] or Angular[11]. Another reason for choosing React was that both team members had prior experience using it.

Material UI is a library built on top of React. It offers a set of predefined components such as buttons, text fields, loading bars and other UI-elements. It implements Googles Material design guidelines[12] using React Components, which simplifies the process of maintaining said guidelines for the entire project. In addition, it allows one to generate CSS within React components, which reduces the total number of files and makes it easier to keep track of the CSS for individual graphical elements.

2.5 Related work

This section highlights research which we deemed relevant to our research goal. The intention of this section is to give an overview of methods which aim to give a better understanding of convolutional networks. We summarize a few papers which contain relevant work that is either important, interesting or both. We also present several papers which explain various visualization methods that we implement or take inspiration from in this thesis. This section gives an impression of the current state of visualizing neural networks, as well as a basic understanding of the methods we implement.

2.5.1 Visualizing and Understanding Convolutional Networks

Throughout the previous years, convolutional neural networks have displayed an impressive performance on tasks such as image classification, and object detection.

[10] https://www.djangoproject.com/
[11] https://angular.io/
[12] https://material.io/guidelines/

(Zeiler and Fergus, 2014) is a paper in which the researchers wish to achieve a better understanding of why convolutional networks do so well, and how to improve their performance even further. They do this by using a system that visualizes features learned in different layers of a convolutional neural network. The paper explains a technique which projects the output of any layer in a convolutional network back to the pixel space. The technique is called Deconvolution(Zeiler, G. W. Taylor, et al., 2011). This method has been used several times(Mohan, 2014; Noh et al., 2015; Stutz, 2014), both for visualization and other purposes such as image segmentation. The idea is to have one layer in the deconvolutional network for each layer in the original network. Each of the new layers will reverse the actions of its neighbor in the convolutional network. In order to analyze the output of one neuron, the output of the other neurons in the layer is set to zero. Afterwards, the deconv-network is run using the feature map of the selected layer. The deconv network will then output an image which shows what triggers the selected activation.

Figure 2.18 shows an example of a convolutional network with its deconvolutional counterpart. The original network consists of three convolution and pooling operations, and the deconvolutional network consists of layers undoing the operations from the original network. Each operation generates a new tensor, denoted as white squares in the figure. The figure does not show the ReLU activations which follow after each convolutional operation, in order to keep it simple.

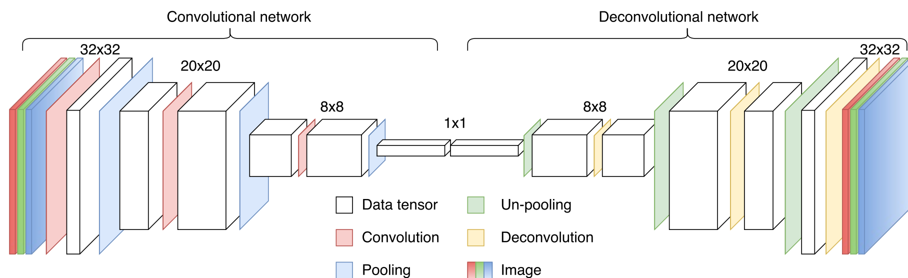


Figure 2.18: Illustration of a deconvolutional network structure, inspired by the illustration in (Noh et al., 2015).

Un-pooling layers reverse the process of max-pooling. When one executes a max-pooling operation many values are lost in the process, as only the highest values are saved in the output of the layer. The solution is to save the spatial location of each value and to set all other values to zero when performing un-pooling.

As we can see in Figure 2.19 the local maximum in each pooling region is preserved. The deconvolutional layer applies a transposed convolution(Dumoulin and Visin, 2016) to its input values. Afterwards the output is run through a ReLU activation-function. In order to create a visualization for one particular filter one has to set all output values not connected to the particular filter to zero, and pass the output into the corresponding layer of the deconvolutional network.

The authors recreate the architecture described in (Krizhevsky et al., 2012), and

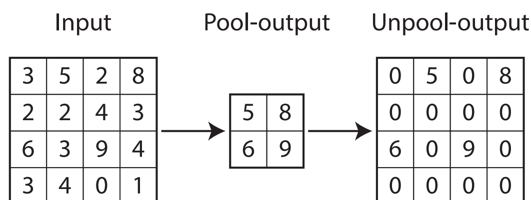


Figure 2.19: Example of an un-pooling performed on a max-pooling operation. The original Max-pool uses a pooling region of 2x2 with a stride of 2.

explain how to improve that architecture by analyzing the visualization of the network. They are able to achieve an increase in accuracy of 1.7% compared to the original network. The visualization showed that the features learned are far from random, and have certain desirable properties such as compositionality, increasing invariance and class discrimination. The paper also showed that the network was sensitive to structure within the input, by occluding certain parts of the input image. Another feat of the paper was showing that the trained network would generalize rather well for usage on other datasets such as Caltech-256(Griffin et al., 2007).

2.5.2 Understanding Neural Networks Through Deep Visualization

This paper(Yosinski et al., 2015) presents two different methods for visualizing and understanding convolutional networks. The first method uses a real-time input to visualize the activation of every filter within a trained convolutional network. The writers of the paper created a system that used a webcam as input for a trained neural network. The system shows the activations of all nodes within the network, sorted by the different layers. This representation makes it easier to understand what kind of input activates the different filters.

The second system which is used for visualization is more similar to backpropagation. The basic concept of the system is to change the input image in such a way that it achieves a higher activation value for a given layer. This in itself is not a new idea, but the paper introduces a set of regularization techniques with which the authors achieve images that are easier to interpret. Earlier attempts to visualize preferred activation patterns often resulted in images which did not resemble any natural images, and therefore did not offer any better understanding of what the network had learned, or how it interpreted images. L2 regularization(explained in Subsection 2.1.6) is introduced as a method for suppressing single high activation values. While such values might increase the activation value, they do not occur in natural images and are also not helpful for visualization.

Gaussian blur is used to penalize areas with high-frequency information. Similar to high-value pixels these areas do not help with the visualization process.

After applying L2 regularization and Gaussian filtering, the image will consist of smaller and smoother values. Two types of clipping methods are introduced to improve the final visualization further. In the first method, clipping with small norms is performed, where a norm is calculated for each pixel, and pixels with small norms are set to zero. An alternative clipping method is clipping pixels with a small contribution. The method approximates how much a single pixel influences the final activation. Pixels with a small influence are set to zero.

2.5.3 Regularization techniques and image priors

In recent years, a plethora of papers have been released (Goodfellow et al., 2014; Mahendran and Vedaldi, 2014, 2015; Wei et al., 2015; Yosinski et al., 2015), describing new inventive ways of improving the feature visualization explained in the section above. These techniques range from performing simple regularization during optimization, such as the blurring and L2 penalization already mentioned, to learning complex image priors in order to achieve visualizations which resemble actual objects to a much greater extent.

Expanding on the concepts introduced in (Yosinski et al., 2015), Google released an article in 2015 titled "Inceptionism"(Alexander Mordvintsev, Christopher Olah and Mike Tyka, 2015). It describes a method they aptly named DeepDream, which gained a lot of fame after producing "dreamed" images of what has later been referred to as algorithmic pareidolia (*DeepDream - Wikipedia* n.d.). Just as in previous feature inversion techniques, the method alters the inputs to a trained CNN in order to maximize activation-values from a given layer. What separated DeepDream from its predecessor(Yosinski et al., 2015), was the fact that they used actual images as inputs. This, in turn, made the optimization process look for features already present within an image, before enhancing them. A bush looking like a dog could turn more dog-like etc. They also introduced a few new regularization-techniques, such as running the algorithm over multiple scales of the image called "octaves", and shifting the gradient over by some pixels, "jittering" it, before applying it to the image.

When we talk about natural image priors, we are taking into consideration certain heuristics, statistical properties or "rules" that are usually present in natural images. These can vary greatly in complexity. When implementing such image priors into the feature visualization process, the priors could either be crafted by hand or learned, by for example extracting some type of information from a dataset of images. Some simple priors, like the fact that sharp noise does not occur in natural images, could be achieved with regularization methods like blurring. By simply maximizing activations without taking into consideration properties of natural images, one might end up with something similar to adversarial images, sort of "fooling" the network(Goodfellow et al., 2014). Examples of handcrafted priors include clipping and/or suppressing large values(Yosinski et al., 2015), blurring(Yosinski et al., 2015), jittering(Mahendran and Vedaldi, 2015), increasing total variation(Mahendran and Vedaldi, 2014), and normalizing gradients(Wei et al.,

2015).

Learned priors, on the other hand, are not simple heuristics, but instead models that have been generated on the basis of some available data. GANs(Nguyen, Dosovitskiy, et al., 2016) (Generative Adversarial Networks), being trained to create natural looking images is an example of one approach to learning such priors. Some papers concerning feature visualization have described simpler learned priors, taking information more directly from the dataset of images that have been used to train the CNN. In the paper (Nguyen, Yosinski, et al., 2016), the authors are trying to visualize possible sub-classes of a trained CNN. Their approach is to first cluster similar images from the dataset from within the same class, adding them together, creating a mean image, before running the feature visualization on top of this image, sort of like in DeepDream. The resulting images look a lot like actual objects, but one could argue that this is a form of cheating. A better example of a learned prior was mentioned in a comprehensive article about feature visualization (Olah et al., 2017), where they exploited correlations between colors in the dataset to further enhance the visualizations.

Learned priors can be powerful tools, but there is a caveat to using some of these methods. We may no longer know what information that comes directly from the network we are trying to visualize, and what information that was just taken from the prior. There needs to be a balance between using the appropriate amount of suitable priors, so we do not end up with adversarial images, but are able to interpret the visualizations correctly. We have intentionally avoided too many learned priors in our platform and instead focused on some really efficient handcrafted ones.

2.5.4 Visualizing and comparing convolutional neural networks

(Yu et al., 2014) aims to give a better understanding of deep convolutional networks. Their method for achieving this goal is by visualizing patches in the internal representation space of the network, and by visualizing the information kept in each layer. In addition, they compare two CNNs with different depth and show the advantage of having a deep network. The first visualization technique uses the activation values to find patches within the original image. A patch in this context is a subsection of an image, that is equal to the selected filter size. The patches are then arranged in a two-dimensional array utilizing t-SNE(Laurens van der Maaten and Geoffrey Hinton, 2008). For visualizing the information kept within the network, the authors use deconvolutional networks(Zeiler, G. W. Taylor, et al., 2011), which are explained in Subsection 2.5.1.

The paper explains the architecture of two networks of different depth. The first network, AlexNet(Krizhevsky et al., 2012) is relatively shallow, having only five convolutional and three fully connected layers. The second network named VGG-16(Simonyan and Zisserman, 2014) has 16 weighted layers, as mentioned in Sub-

section 2.3.1. In addition to depth variations, the filter size and stride also differ between the networks.

The comparison of the two networks was done by looking for differences within the visual representations. The information extracted by the deconvolutional networks showed that insignificant features within the image were gradually removed, while discriminant features would stand out more. The results show that the more shallow CNN retains more irrelevant information than its deeper counterpart. Shallow in this context refers to having fewer layers and parameters. The AlexNet is referred to as shallow while the VGG-16 network is referred to as deep. In addition, the deeper network would have a higher level of sparsity within its high-level layers. The sparsity was measured by the proportion of zero activations of a layer.

2.5.5 The LRP Toolbox for Artificial Neural Networks

When used in the context of classifying images with convolutional networks, layer-wise relevance propagation(Bach et al., 2015) or LRP for short is a method for tracing a prediction back to the input pixels. In other words, the method shows to which degree each input pixels contributed to the final prediction. The contributed amount in this context is also called relevance. This method is very similar to Deep Taylor Decomposition(Montavon et al., 2017) and can be seen as an alternative, which is also discussed in (Bach et al., 2015). The main difference between these two techniques is that Deep Taylor Decomposition linearly approximates the contribution of each pixel while LRP applies a propagation rule in order to propagate the relevance from the output of the network to the input pixels. Both methods satisfy the constraints for heatmapping described in Subsection 2.5.6.

The LRP Toolbox(Lapuschkin et al., 2016) provides an implementation of the LRP algorithm in Python and Matlab. The goal was to be able to familiarize users with the algorithm, and to explain the prediction of pre-trained networks. A demo is provided in the form of a website[13] which retrieves the heatmap of a user-selected image. The user has the possibility to choose between different LRP-formulas and adjust the parameters of the selected formula.

Figure 2.20 shows an example of relevance propagation. The original network has three input variables A, B and C, and one output node. A set of input values is fed into the network, which results in a value in the output node. This value is then propagated backwards through the network, and each node is assigned a value, based on its contribution to the final output. The sum of values in each layer equals the value in the output layer. Each output node in the LRP graph corresponds to one input node (X to A, Y to B, etc.). As we can see from the example the value of the variable C had the highest contribution to the final output value of the original neural network.

[13] `https://lrpserver.hhi.fraunhofer.de/image-classification`

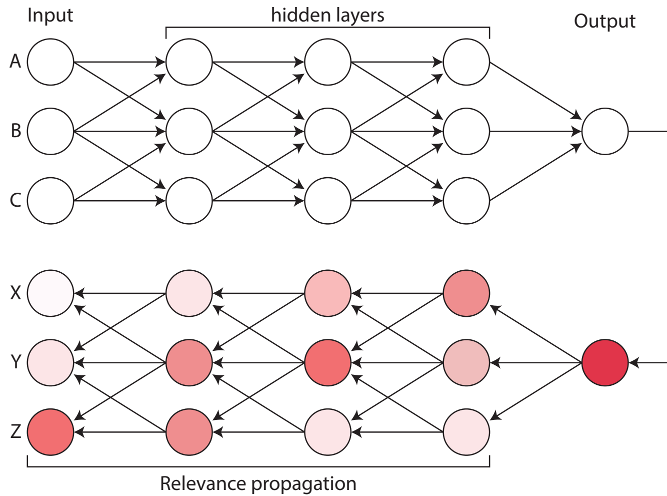


Figure 2.20: Example of relevance propagation in an ordinary neural network.

2.5.6 Explaining NonLinear Classification Decisions with Deep Taylor Decomposition

This paper(Montavon et al., 2017) explains how to apply Taylor Decomposition in order to achieve a better understanding of the classification process happening within convolutional neural networks. The method exploits the first order Taylor expansion to create a formula for approximating the contribution of each input pixel to the final classification. The Taylor series expresses a function as a sum of its derivatives at some point a. First order Taylor expansion, as the name implies only uses the first order derivative to express the function.

The final distribution of relevance upon the input pixels can also be referred to as a heatmap. A heatmap is a visual representation of data, where different data values correspond to different colors. Since each pixel in the original image has a corresponding value in the relevance heatmap, the relevance scores can also be visualized as an image. In order to create a heatmap, two constraints have to be satisfied.

First, a heatmap has to be conservative. This constraint implies that the sum of all relevances in one layer has to be equal to the total amount of relevance found. For a network utilizing a softmax function, the sum of all output nodes equals one. In other words, a network using softmax as its final layer should have a relevance which sums up to one, in all hidden layers.

Another constraint is that all values forming the heatmap have to be greater than or equal to zero. With this constraint the heatmap will only be zero in the absence of an object to be detected, i.e. the heatmap would display zero relevance for all pixels. Without this constraint, the heatmap would also display zero if there was as much

positive relevance as there was negative relevance. The LRP function described in Subsection 2.5.5 is another example, besides Deep Taylor Decomposition, which satisfies these constraints.

By utilizing the first order Taylor expansion, the authors of the paper establish a set of formulas which propagate the relevance backwards one layer at a time. The different formulas correspond to different constraints on the input parameters. For convolutional networks utilizing the ReLU-activation function, two formulas are relevant. The equations calculate the Relevance R for a node i using the weights between the two layers, w, the activations x from the layer which i is a part of, and the relevance from the previous layer R_j.

z^+-Rule

When using ReLU activation functions, the output of one layer is restricted to $X \subset \mathbb{R}_+^d$. As the layers are chained together the output of one layer becomes the input of the next. Therefore we can say that the input of a convolutional layer holds the same restriction. The accompanying relevance propagation rule is expressed in Equation 2.12.

$$R_i = \sum_j \frac{z_{ij}^+}{\sum_{i'} z_{i'j}^+} * R_j \tag{2.12}$$

In Equation 2.12 $z_{ij}^+ = x_i w_{ij}^+$ where $w_{ij}^+ = max(0, w_{ij})$. Also, $\sum_{i'} z_{i'j}^+$ equals the sum of all activations going in to node j.

$z^{\mathcal{B}}$-Rule

The first layer of the neural network has a stricter constraint on its input than the other layers, as it takes in the original image. The separate pixel values are restricted to be within a certain range. This creates a new Equation for propagating the relevance.

$$R_i = \sum_j \frac{z_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-}{\sum_{i'} z_{i'j} - l_{i'} w_{i'j}^+ - h_{i'} w_{i'j}^-} * R_j \tag{2.13}$$

In this equation $z_{ij}^+ = x_i w_{ij}$, $w_{ij}^+ = max(0, w_{ij})$ and $w_{ij}^- = min(0, w_{ij})$. l_i and h_i are equal to the lowest and highest pixel values for the selected dimension. One dimension in this context refers to one color channel in the RGB color space. l_i and h_i hold the following constraint: $l_i \leq 0 \leq h_i$. The elements $-l_i w_{ij}^+$ and $-h_i w_{ij}^-$ help maintaining a positive value for the fraction in the equation and restrict the relevance to $R_i \geq 0$

Utilizing matrix operations

In order to efficiently calculate the relevance for all nodes in a layer, matrix operations are used. Algorithm 1 corresponds to Equation 2.12 while Algorithm 2 corresponds to Equation 2.13. Both algorithms are taken from the Appendix of (Montavon et al., 2017)

Algorithm 1 z^+-Rule

Input:
Weight matrix $\mathtt{W} = \{w_{ij}\}$
Input activations $\mathtt{X} = \{x_i\}$
Upper-layer relevance vector $\mathtt{R} = \{R_j\}$

Procedure:
$\mathtt{V} \leftarrow \mathtt{W}^+$
$\mathtt{Z} \leftarrow \mathtt{V}^\top \mathtt{X}$
return $\mathtt{X} \odot (\mathtt{V} \cdot (\mathtt{R} \oslash \mathtt{Z}))$

Algorithm 2 $\mathrm{z}^{\mathcal{B}}$-Rule

Input:
Weight matrix $\mathtt{W} = \{w_{ij}\}$
Input activations $\mathtt{X} = \{x_i\}$
Upper-layer relevance vector $\mathtt{R} = \{R_j\}$
Lower-bound $\mathtt{L} = \{l_i\}$
Upper-bound $\mathtt{H} = \{h_i\}$

Procedure:
$\mathtt{U} \leftarrow \mathtt{W}^-$
$\mathtt{V} \leftarrow \mathtt{W}^+$
$\mathtt{N} \leftarrow \mathtt{R} \oslash (\mathtt{W}^\top \mathtt{X} - \mathtt{V}^\top \mathtt{L} - \mathtt{U}^\top \mathtt{H})$
return $\mathtt{X} \odot (\mathtt{W} \cdot \mathtt{N}) - \mathtt{L} \odot (\mathtt{V} \cdot \mathtt{N}) - \mathtt{H} \odot (\mathtt{U} \cdot \mathtt{N})$

$\odot$ stands for element-wise multiplication, while $\oslash$ denotes element-wise division. Both of these algorithms satisfy the heatmap constraints mentioned earlier. The total amount of relevance is preserved from one layer to the next one, which complies with the conservative constraint. Algorithm 1 only uses the positive part of the weights, while Algorithm 2 constrains its values to be positive or zero, thereby satisfying the positive constraint.

Chapter 3

Method

In this chapter, we explain several visualization techniques which we have implemented. Each method originates from articles or papers presented in Section 2.5. We also give a short explanation of how we designed our platform for experimenting with different networks and visualization techniques.

3.1 Application design

The core application was implemented following the client-server architecture. This architecture divides the structure of the system into a service provider, known as the server, and one or several service requesters, known as clients. The server offers several services or resources which the clients can take advantage of. Predicting the class of a given image is an example of a service. In order to let users interact with the services provided one implements a user interface which acts as a mediator between the user and the server, and displays the resources provided by the server to the user in an understandable and intuitive way. In this project, we refer to the server side as the backend, while the client side is called the frontend.

Figure 3.1 presents the data flow within the application. The server receives a request which is processed and yields a call to the model. The model processes the request from the server, and grants a response, which the server sends in return to the client.

Figure 3.1: Illustration of the dataflow within the application

3.1.1 Backend

The backend core was implemented in Flask. It is modeled after the REST-API architecture. The server has a set of endpoints, which the client can access using HTTP-requests. The client requests a certain resource of service using HTTP. The server makes use of two types of requests; GET and POST. The client can use a GET request to ask for a resource. If the client needs to submit certain data along with the request, this can be done using a POST request. An example of a POST request could be submitting an image which is to be classified.

Each endpoint returns a JSON-object. This simplifies the transition from backend to frontend, as the frontend is written in JavaScript, which excels at handling JSON-objects.

3.1.2 Frontend

The user interface is implemented using React in combination with Material-UI. The frontend uses two main functions for interacting with the server's resources. One function performs a GET request, and the other performs a POST request, as explained earlier. These functions are implemented as asynchronous operations as one does not wish to lock up the entire user-interface while waiting for the response to a request. A global state is stored within the user interface, in order to preserve the state of the application throughout a session. Reloading the user interface will reset the state. The frontend is explained further in Section 3.6 where we elaborate on the functionalities and our thoughts behind them.

3.2 Feature visualization

This section will go through the techniques we have implemented as part of the feature inversion functionality of the visualization platform. "Features" in this context is just a word used to describe the concepts and patterns (of varying complexity) that the hidden layers inside convolutional neural networks learn during training. A more accurate description of what a feature actually is can be found under Section 1.1. Feature inversion is a technique with similarities to the backpropagation algorithm explained in Subsection 2.1.4, but can be used to produce images depicting approximations of features. There exist many different tricks to improve

the inversion process. Several of these are presented throughout Subsection 2.5.3. After considering all current techniques, we have aimed to achieve state-of-the-art results, both in regards to obtaining natural looking, accurate feature visualizations, in addition to keeping the time spent as low as possible. Choosing another domain for the optimization process, as described in Subsection 3.2.4 and 3.2.5 seemed to make an especially significant improvement.

3.2.1 The naive approach

The simplest "naive" approach to implementing feature inversion is very similar to backpropagation which is carried out during each step when training a neural network. The difference lies in the fact that instead of adjusting weights inside the network, we instead compute how much we need to change the values of the inputs in order to minimize some loss function. Examples of such loss functions, based on activation values from within hidden layers of the network can be seen in Equation 3.1 and 3.2. Although there are better alternatives to this naive approach, we implemented it our platform in order to get a baseline for judging the quality of visualizations. In addition, we wanted to give users full control over the inversion process, to allow for experimentation and for the sake of scientific curiosity.

In the first step of the naive approach, the input into the network will be an array representing an image, filled with random values within some normal distribution. Including randomness in the initial image, makes it easier for the optimization process to form features and patterns, compared to starting with an image consisting of only one color. The initial input could also be an actual image. This is what Google did with their "DeepDream" technique, which gained fame after they created a blog post explaining the steps of the algorithm(Alexander Mordvintsev, Christopher Olah and Mike Tyka, 2015). Using an actual image, the patterns that are already present in the image will then be enhanced throughout the optimization with regards to the layer chosen. The resulting optimized image when using the same method as in DeepDream is often referred to as the "dreamed" image.

After loading in a trained neural network and the random image array which will be used as input, we can start the backwards propagation step of computing gradients. The process of computing gradients is made much simpler thanks to the built-in functions of Tensorflow. First, the tensor for the output-layer whose activations will be maximized during the run is selected. After choosing the layer, we can also pick out one or more specific channels (filters) from this layer, or try to maximize the output for the entire layer. When a single channel is used, the loss function will be based on the mean value across all the activations from the two spatial dimensions, which also can be seen as the x and y dimensions. When maximizing for more than just a single channel, we can use the same mean as before for every channel, and then add them together to find the mean across all of these loss functions. Given that the selected output-layer is a standard convolutional layer, the following expressions show the steps to deriving loss-functions that can be minimized during feature inversion.

The expression below shows a single activation value, as defined in Equation 2.6 (notation remains the same):

$$O_{i,j,l} = (\sum_{z=0}^{d-1}\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} I_{x+a,y+b,z} * F_{x,y,z,l}) + B_l \tag{3.1}$$

For a single channel, where t represents the index of the input and r is the total number of inputs, the loss function can be written as:

$$Loss_c = -(\frac{1}{i * j * l * r}\sum_{t=0}^{r-1}(O_{i,j,l})_t) \tag{3.2}$$

The expression inside the outer parenthesis describes the mean output across one channel. This activation value is supposed to be maximized, which is why we negate it in order to turn it into a loss function.

For an entire layer, where c represents the channel number, and q is the total number of channels in the layer, the loss function can be written as:

$$Loss = \frac{1}{q}\sum_{c=0}^{q-1} Loss_c \tag{3.3}$$

One of these Loss-functions can then be used to compute the gradient $(\Delta\theta)$:

$$\Delta\theta = \frac{\partial Loss}{\partial Input} \tag{3.4}$$

The Input variable in Equation 3.4 refers to the actual pixel values of the image we are optimizing. To compute the gradient in practice, we first insert a tensor that is equivalent to one of the cost functions above, in addition to the input tensor itself into the gradients-function supplied by Tensorflow. The function will return the computational graph for the derivative of the loss-function with respect to the input tensor. This graph, together with actual input data, will be used in a Tensorflow session whenever we need to compute the gradient. The exact form of the loss functions will be slightly different if the layer or collection of neurons we want to maximize is of a different type. This feature inversion process is not limited to just convolutional layers, but can also be applied to all kinds of other layers, such as max-pooling or fully connected layers.

Before applying the gradient directly to the image, it can be useful to first normalize the gradient by dividing it by its standard deviation. This avoids very large or small values in the gradient and ensures a smoother gradient ascent. Finally, the gradient is multiplied with the step size (η) and added to the pixel values of the input tensor. This whole process is run over many iterations to create the final optimized/dreamed image.

$$Input_{t+1} = Input_t + \eta * \Delta\theta_t \tag{3.5}$$

3.2.2 Transformations

When we are training convolutional neural networks on datasets of images, as described in Subsection 2.1.6, different transformations techniques are some of the simplest forms of regularization methods that can be applied. Even though they are quite simple, they can improve classification by a wide margin (L. Taylor and Nitschke, 2017). In order to make convolutional neural networks more robust and better at recognizing variations of the same object, it is normal to perform various transformations on the images themselves before feeding them into the network. These types of transformations include scaling the images, rotating them, cropping them, slightly changing the colors and so forth. Some built-in randomness in this process is also beneficial. Since these methods are used when training CNN's it makes sense to use similar techniques when trying to achieve the most accurate depictions of features, during feature inversion.

Using simple transformations can have a huge impact when it comes to achieving beautiful images, where the actual feature we are trying to visualize is easier to recognize. One way they can be included in the aforementioned feature inversion process is to transform the gradient before applying it to the optimized image. Another manner of approaching the problem is by transforming the optimized image itself before computing the gradient and then applying this gradient to the same transformed image. The second method is used in our implementation because it fits nicely with the rest of the TensorFlow structure described in Subsection 3.2.3.

Jitter

The transformation that usually made the largest impact on quality during optimizing is referred to as jitter, a term used by Google in their Deep Dream implementation[1]. By "jittering" an image we mean to shift it over by a set number of pixels, so the dimensions stay the same, but the start of the image might now reside on the other side of where it was originally. An example of an image that has been "jittered" along the x-axis by -15 pixels can be seen in Figure 3.2.

Figure 3.2: Example of the jitter-transformation applied in the x-direction.

Performing this operation on an image-array can be done using the *rollover* function

[1] `https://github.com/google/deepdream`

in NumPy[2], a Python library. We wanted to execute it using a Tensorflow tensor instead, but unfortunately, there does not exist any equivalent to the rollover yet in the Tensorflow library. There does exist a random-crop tensor, however, which turned out to serve much of the same purpose. The most important aspect of this transformation is that it stops the gradient from starting at the same place every time, which will, in turn, saturate single pixels and certain elements of the feature in the optimized image too much. An example of the difference jittering by some random number between 1 and 10 makes, can be seen in Figure 4.8 under the results presented in Subsection 4.1.2.

Scaling

Another useful transformation that has been implemented is scaling of the optimized image, by some random factor. This random factor should not be far above or below 1. With a scaling factor that is either too large or small, the resulting gradients will turn out a bit poor, due to the changes in resolution. The scaling method we have used is bilinear and is performed with a Tensorflow tensor. An example with and without scaling is displayed under results in Figure 4.10. When applying scaling during feature inversion, we might end up with an image that has enhanced feature(s) of many different sizes. Depending on the layer chosen, we might also end up seeing patterns within patterns.

Rotation

While jittering the optimized image helps reducing noise and removing unwanted artifacts, rotating the image by a random angle within a specified range may be even more useful in order to get accurate feature-depictions. When the image is optimized at exactly the same angle every time, depending on the filter, there is a risk that the optimization will get stuck on a repeating pattern. Allowing the image to be rotated too much, however, will result in various degrees of cyclic symmetry within the emerging patterns. These resulting images might look pretty but are often missing aspects of the actual features. Figure 4.9 shows the impact the rotation transformation has during optimization.

Padding

The last transformation we have implemented into the system pads the image. The tensor used in this operation adds n grey pixels in the x and y-direction. Padding the image serves only one purpose. With sufficient space around the image, we avoid some of the artifacts that might occur at the very edges of the image. These artifacts say very little about the feature in question, since the straight lines at the edges, which affect the gradient during optimization, rarely occur in natural

[2] `http://www.numpy.org/`

images. The number of pixels that we pad should not exceed the minimum required amount by much. It is important to remember that increasing the input dimensions will also exponentially increase the gradient computation time.

3.2.3 Utilizing TensorFlow for faster visualization

Executing mathematical operations with the help of TensorFlow tensors can be a great tool for speeding up computation time. This holds especially true if the system has powerful GPUs available, which are designed to run calculations in parallel. A more in-depth description of TensorFlow can be found under Subsection 2.4.1.

Because the input is just a collection of parameters we are trying to optimize towards an objective, instead of feeding slightly different input into the network each time (see Subsection 3.2.1), representing the input as a trainable variable-tensor makes a lot more sense and speeds up the entire process. With the rest of the weights in the graph frozen, the optimization process can be carried out by simply training the network. Considering this, we realized that the entire feature-visualization actually could be accomplished through a single TensorFlow graph. The template graph we ended up with consists of four interchangeable sub-graphs, which are connected in sequence as seen in Figure 3.3

The first graph, which contains the only trainable tensors, represents the parameter space we are trying to optimize. In the naive implementation, this would be a single trainable tensor with the shape: (width, height, color channels). The output of this sub-graph is then used as the input to the next, which handles all of the transformations mentioned in the previous subsections. This graph uses tensors with built-in randomness to apply all of the image-transformations that we want to include. The output of the transformation graph is subsequently used as the input into the actual convolutional network. This is the network that has been trained on a dataset of images and is the part of the graph that contains the actual information that we are trying to extract and visualize. From one or more selected tensors within the pre-trained network, we connect one last sub-graph, which is used to compute the desired loss function. This loss function is used as the input to a chosen optimizer. Running the optimizer-tensor in a Tensorflow-session for one step will optimize the parameter space in the input by a small amount, depending on the optimizer's learning rate. The entire flow of the visualization graph can be seen in figure Figure 3.3. In our implementation, we have created a class named *graph_builder*, which sets up the two first sub-graphs according to a wide array of parameters.

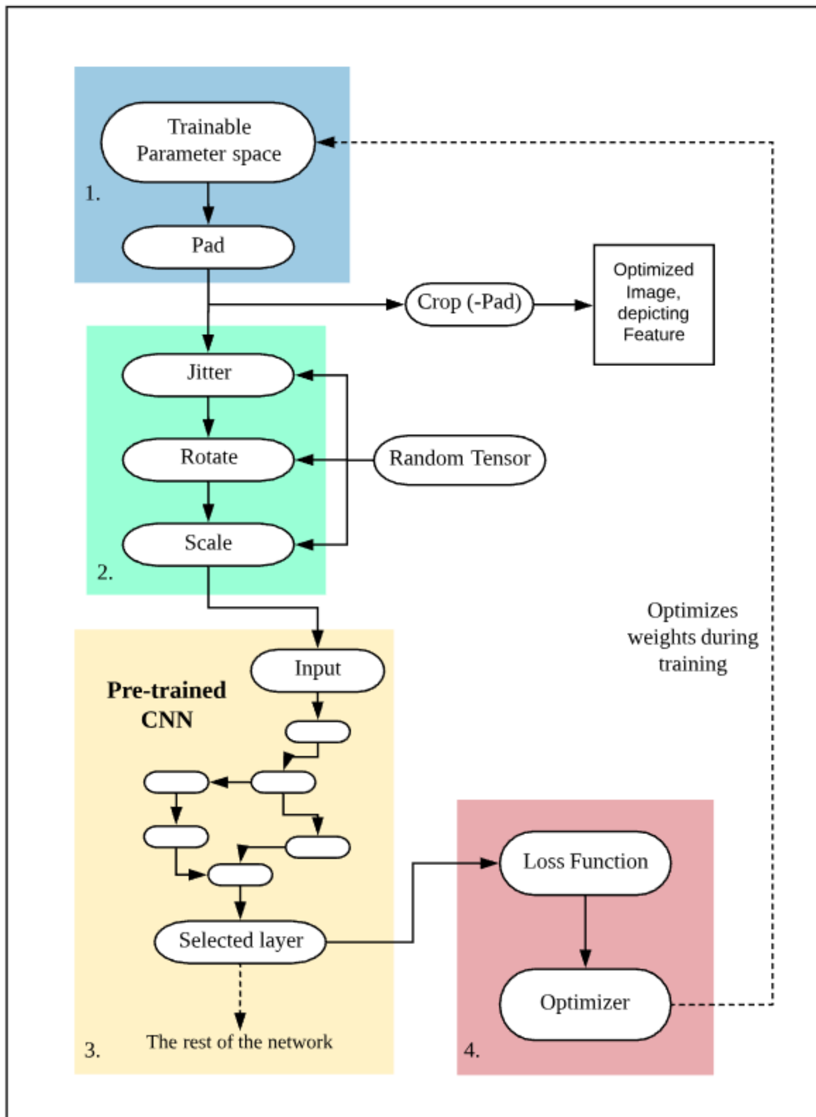


Figure 3.3: The four interchangable sub-graphs, making up the Tensorflow graph structure used in feature inversion.

3.2.4 Alternative parametrization spaces

The most straightforward, naive way to do feature inversion as described in Subsection 3.2.1 is not the only way to optimize an image. Instead of looking at the RGB-values of the pixels themselves as parameters we want to change, we can in-

stead try to look for alternative parameterization spaces. There exist many other commonly used spaces we can utilize to represent an image. These spaces are often used in various image-filtering processes. An image represented in the RGB-space is first transformed into the new parameter space, e.g., the Fourier space. Some of the values are altered to achieve the desired filter effect, and finally, the inverse of the previous transform is applied to go back again to an image representation in the standard RGB-space.

We only perform the last two steps during our feature inversion. The first tensor of our graph represents the alternative parameter space, which will be the only trainable variables in the Tensorflow-graph. The next step is to transform it through Tensorflow-tensors into a RGB-image, which is then used as the input to the transformation sub-graph. A small change in one parameter space can have a significant impact on the resulting image we end up with after we perform the transformation into the RGB-space. This can speed up the optimization considerably.

Optimizing in the Fourier space

One of the parameter spaces that resulted in the most natural-looking visualizations and fastest training times during feature inversion turned out to be the Fourier space. It is often referred to as the frequency domain due to the nature of the Fourier transformation. The standard way to represent images, with pixel-values, is called the spatial domain. Instead of representing images by the pixel values in the spatial domain directly, we look at the rate at which the pixel values are changing within the spatial domain, hence the frequency term. The FFT (Fast Fourier Transform) algorithm is used to convert between the domains because of the significant improvement in speed $\theta(n * log(n))$, compared to directly using the DFT (Discrete Fourier Transformation) algorithm, which runs in $\theta(n^2)$ for n points.

Before the feature inversion can start, we first need to initialize the frequency domain with random values within a small normal distribution. In order to know the appropriate dimensions for this space, it is important to know some of the details as to how the 2D-FFT algorithm is computed. It works by first taking the FFT of one row at a time, which results in an intermediate array of the same size. We then take the FFT over all the columns of this new array. We only need to compute half of the values of each column, due to the symmetric properties of the FFT(Guo et al., 1998). In case of an odd number of elements inside each column, half means taking the floor division of elements plus one.

The dimensions of the parameter space are thus set to be (2, 3, *r_freq, c_freq*), where the two last dimensions represent the frequencies over the rows and columns of the image. If we want the resulting image to be of dimensions (n, m), then r_$freq = n$, while $c_freq = floor(m) + 1$. The second dimension is of size 3, since we run the Fourier operation over each color channel separately, while the first dimension size is 2, because each point in the frequency domain is made up of a

real and an imaginary part.

After initializing the parameter space, the real and imaginary sub-arrays are combined into a single array of complex numbers. We also have to scale this array by 1/frequency-bins, in order to get values to be optimized correctly. The 2D array of frequency bins is computed by taking the frequency bins in both directions using NumPys *fftfreq* function and combining them using Euclidean distance: $\sqrt{r_bins^2 + c_bins^2}$. After properly scaling the complex array, it is fed into an inverse 2D-FFT tensor, from the Tensorflow library. The equation for doing the inverse 2D fast Fourier transformation is shown in the formula below:

$$f(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} F(u,v) e^{j2\pi(\frac{ux}{m} + \frac{vy}{n})} \tag{3.6}$$

F(u, v) is the notation used for the transformation into the Fourier space, so in this context it denotes a single point in the randomly initialized frequency domain described above. m and n represent spatial dimensions, while j is the imaginary unit.

Optimizing in the Laplacian pyramid space

Another promising parameter space that was implemented into the platform is based on Laplacian pyramids. This is yet another way to represent an image, where a small change in a single parameter can make a big difference in the actual image. In a pyramid representation of an image, the original image is blurred and downsampled n times, until we end up with n smaller versions of the original image. Stacking them on top of each other in a visual representation of the space will create a pyramid with n levels, hence the name. In the case of Laplacian pyramids, the differences between each level are saved, to be used for various applications such as image compression. It is these kinds of arrays, containing the differences on each level, that make up the parameter space we are optimizing during the feature inversion.

In order to initialize this parameter space, we start out by creating a tensor with the desired dimensions of the final optimized image. This tensor will be one of n trainable variables and is initialized with random values from a normal distribution. For the next iteration, we create a new variable tensor with the spatial dimensions being half the size of the spatial dimensions of the previous tensor. The dimensions for iteration n is in other words set to be the original dimension sizes divided by 2^n. It is initialized with random values, before being scaled up to the dimension of the previous tensor, using bilinear upsampling. This process of creating a smaller tensor, initializing it and upsampling it is repeated until we have gone through all n levels.

Every time we create a new tensor, we also add it to an overarching pyramid-tensor, which will contain the sum of all the (upsampled) tensors from every level. This

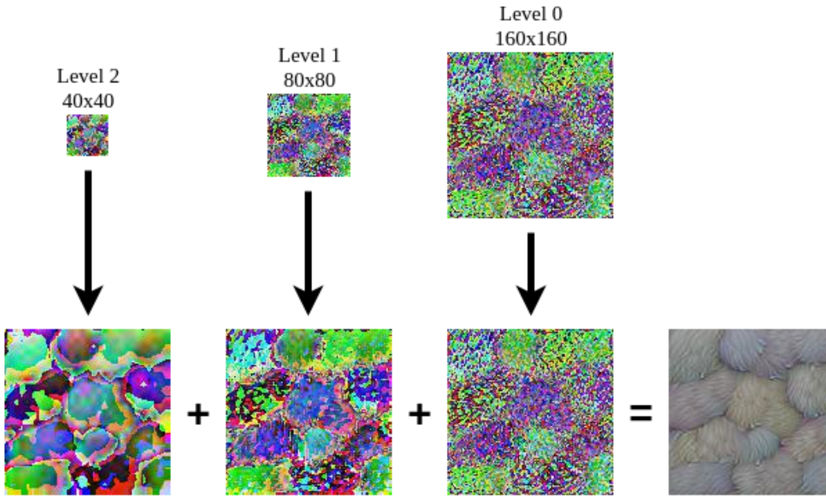


Figure 3.4: All the levels of a Laplacian pyramid with n = 3 are upsampled and combined into the optimized image on the right. Optimization objective chosen for this example was channel: 134 from layer: mixed_4c in the Inception net.

is just a simple add operation. Since all these tensors represent the differences between every sub-sampling in a Laplacian pyramid, adding them all together should naturally result in the "original" high-resolution image. The Figure 3.4 shows three levels added together, during the beginning of an actual run. Small changes in pixels of low-resolution levels will naturally change many pixels in the high-resolution image. Because of this, the optimization can be sped up considerably. One thing to notice is that if we only use one layer, this parameter space becomes equivalent to just optimizing using the naive approach.

3.2.5 Decorrelating color channels

Colors within natural images are often correlated. What we mean by this is that if we plotted all the RGB-values for each pixel in a photo onto a 3D space, a lot of points would most likely fall somewhere along a diagonal line from (0, 0, 0) to (255, 255, 255). This is partly caused by the fact that light and darkness are common attributes of natural images, causing RGB-values to not stray too far from each other. More complex correlations between colors in natural images exist as well.

When training a neural network on a set of data, it is usually not optimal to have inputs that are correlated. If there exist correlations between input variables, these can be learned by a network during training and ultimately be represented by combinations of weights inside hidden layers. This encoding might be completely redundant though if the correlations are known beforehand. In other words, if we

already know correlations that exist between variables, we should probably take advantage of this knowledge before feeding the data into the network. This is called decorrelating. By first measuring the explicit correlations between variable pairs in the training data, we can compute a symmetrical covariance matrix. After performing a single value decomposition on this matrix, we end up with three matrices: $U\Sigma V^*$, where U represents eigenvectors. Multiplying the input data with U will decorrelate the results.

A trick we can apply to generate even more natural looking, faster feature visualizations is to decorrelate the color-channels. This can be achieved by using a learned prior, in this case, a 3x3 covariance matrix based on the colors within the Imagenet dataset(J. Deng et al., 2009), which the CNNs we use as examples (see Subsection 2.3.2 and 2.3.1) have been trained on. Multiplication of U is performed with a tensor within the parameterization sub-graph from Figure 3.3. An important aspect to note when decorrelating input variables is that the basin of attraction during training should not really move too much, but instead change its shape. What this means is that the training can speed up considerably, but the "most correct" model should preferably still remain the same. In practice, when performing feature visualization with decorrelation of colors, we achieve natural looking images with less saturated colors faster than if we did not decorrelate. This can be seen under Subsection 4.1.3 of results. In theory, both approaches should hopefully be able to converge towards the same objective with enough training(Halkjær and Winther, 1997).

Even though the implementation of this color decorrelation is hard-coded into the platform, with correlation values extracted directly from the Imagenet dataset, it should probably work well for most other CNNs trained on natural images. This holds true, assuming Imagenet contains enough images, with enough variety and only negligible biases concerning the colors within images contained in the dataset.

DeepDream

In the special case of DeepDream, where we run feature-visualization over an actual image, we were able to take advantage of the decorrelation described above. The resulting dreamed images using this technique ended up looking a lot better, having not altered the underlying colors of the original image too much. This can be seen under results in Subsection 4.1.5. To implement decorrelation in DeepDream, we need to first multiply the input image with the inverse: U^{-1}, before using these values to initialize the parameterization space. This parameterization space will be multiplied by U afterwards, as described previously.

3.2.6 Training the neural network

In earlier sections, we have described the implementation of the first three subgraphs, handling the parameterization space (3.2.4), the transformations (3.2.2) and the pre-trained network (2.3), but it is the last part that actually trains the network during visualization. The loss function in the final implementation, which is derived from a linear combination of layers and channels is still computed the same way as explained in the naive implementation in Subsection 3.2.1. This loss function is then passed to an optimizer tensor provided by the TensorFlow library. After testing out most of the built-in optimizers, we chose Adam as the default, since it produced both fast and pretty feature-visualizations. An explanation of how Adam works can be found in the background Subsection 2.1.5.

In the main function that takes care of the feature inversion, the optimizer is one of the passable parameters, for easy configuration. The learning rate passed as a parameter to the optimizer is also an important factor in the training process. Depending on the optimizer and the chosen parameterization of the input space, the appropriate learning rate differs a lot as well.

We made sure it was possible to pass an entire list of n different optimization objectives into the visualization function at once, in case images of multiple different feature inversions are required at the same time. In order to save time between each separate feature inversion, only the minimal required steps are taken between each of these runs. This includes the initialization of the trainable parameter space values. If Adam is used as optimizer, we also need to retrieve and initialize some of its own beta-parameters, due to a bug in TensorFlow[3]. A new loss function, based on the n'th optimization objective will also need to be created at the start of each run.

3.2.7 Multiple optimization objectives

Up until now, we have mostly described optimization in relation to a single objective. Either trying to maximize the output of an entire layer or a specific channel inside a layer. The loss function which drives the optimization can, however, include an unlimited amount of different objectives based on every little part that makes up the network. After selecting multiple objectives, they can be freely combined in any kind of linear fashion. If someone wants to know how two different filters would interact with each other, they can add the two objectives together as seen in Equation 3.7.

$$Loss = \sum_{n=0}^{N-1} \alpha_n Loss_{obj_n} \tag{3.7}$$

[3] `https://github.com/tensorflow/tensorflow/issues/8057`

The alpha factor can be used to put a larger emphasis on one objective over another. Some filters are slightly faster to optimize than others, so changing this parameter is a way to produce images that capture the essential elements from multiple objectives.

The way feature-mixing has been implemented is similar to the way multiple visualizations has been implemented in the system. By sending a list of objectives (but now also coupled with alpha values) into the visualization function, they will automatically be combined into a single loss.

3.2.8 Fine-tuning parameters

In our final implementation of the visualization platform, we made sure that it was easy for users to alter and fine-tune the most relevant parameters that affect the results of the feature inversion process. A listing of all these parameters can be found under Subsection 3.6.3, which also gives a description of the graphical user interface. We wanted to offer users all this freedom for multiple reasons. There exist a lot of uncertain factors that can play a role during the training process described in the previous paragraph. The structure of the trained convolutional network, magnitude of weights and so forth can play a part in finding the optimal settings, which is why they are available to modify. However, the settings that were found to generally produce the best results in reasonable time, with the example CNNs, are selected by default. There is also a trade-off between making optimized images that are pleasing to the eye and resemble actual objects, vs. finding the actual “raw” feature that the network has learned. This concept is elaborated on in Section 4.1. Another important trade-off is time usage. Adding extra transformation tensors, lowering the learning rate and using a slow parameter space for optimization will slow the process down considerably.

One last reason for giving users the freedom to set most of the parameters is to facilitate the creation of artistic images. This aspect has not been a big focus throughout this project, but is more of a byproduct of the underlying tools. Especially with the DeepDream component, it is possible to create dreamed images with a lot of artistic merit.

3.2.9 Fetching image-examples of features

When looking at the result of a feature inversion, it might not always be that easy to decipher what feature the generated image is supposed to represent. A simple way to make it a little clearer can be to fetch actual examples of real images that maximize the same neurons as the feature in question. The way we implemented this was by feeding a lot of small images into the CNN during a TensorFlow session. Using the same loss function as we would in feature inversion, we can get a loss value for each image that we fed into the network and rank the images accordingly.

The top ten images with the lowest loss values can be computed and displayed next to a feature-visualization in the platform with the push of a button.

The dataset of images we have used for this feature was taken from the " ImageNet Large Scale Visual Recognition Challenge 2017"[4]. We used the test-set which contains 5500 images of various sizes. n patches of size 64x64 were extracted at random locations from each image. These patches are the small images we feed into the network and give scores depending on how well they represent the feature. Even though there exist image datasets where every image is 64x64 and part of a class, we chose to perform this random cropping on larger images instead, since features are likely to appear as part of an image, rather than appearing throughout an entire image representing a class.

3.3 Activation visualization

As mentioned in Subsection 2.2.2 the output, also known as the activations of a convolutional layer can be displayed graphically. The output can be seen as a set of two-dimensional arrays, each one corresponding to a filter in the convolutional layer. By constraining the values of all two-dimensional arrays to be in the range [0,255], we can display them as gray-scale images. This method is expressed in Algorithm 3.

Algorithm 3 Constraining activation values

Input:

Convolutional activation $\mathtt{Y} = \{y_{ij}\}$

Procedure:

$\mathtt{A} \leftarrow \frac{x}{max(\mathtt{Y})}\mathtt{Y}$

return $255 * \mathtt{A}$

$max()$ returns the maximum value of a given matrix. The returned matrix can then be displayed as an image using an image library for Python.

3.4 Deep Taylor decomposition

This section gives an explanation of how we implemented Deep Taylor Decomposition with Tensorflow. As each layer in a convolutional neural network is unique, so is the propagated relevance. In order to propagate the relevance backwards through a convolutional network, one has to construct a reverse network using the functions described in Subsection 2.5.6. This process is similar to that of a deconvolutional network(Zeiler and Fergus, 2014) since each layer in the original network

[4] `http://image-net.org/challenges/LSVRC/2017/download-images-1p39.php`

will have a corresponding layer in the Deep Taylor network. The simplest way to generate a Deep Taylor graph is to iterate backwards through the network and add a new layer to the new graph, for each new layer encountered in the original CNN. Iterating backwards through a single network was a relatively simple task, as one would know the order in which the layers would appear. Creating a general method, however, proved to be more of a challenge than first anticipated. This problem and its solution are described more thoroughly in Subsection 3.4.3.

3.4.1 Generating a Deep Taylor graph

In order to generate a Deep Taylor graph, we needed to create opposing layers for all different types of layers found within a CNN. The opposing layer to a fully connected layer is created by implementing Algorithm 1 from Subsection 2.5.6. The sharing of weights in a convolutional layer made the backpropagation a bit more tricky. We used Algorithm 4, which is an extension of Algorithm 1 in order to propagate backwards through convolutional layers.

Algorithm 4 Relevance propagation in convolutional layers

Input:
Weight matrix $\mathtt{W} = \{w_{ij}\}$
Input activations $\mathtt{X} = \{x_i\}$
Upper-layer relevance vector $\mathtt{R} = \{R_j\}$
Epsilon $\epsilon = 10^{-10}$

Procedure:
$\mathtt{V} \leftarrow \mathtt{W}^{+}$
$\mathtt{Z} \leftarrow Conv(\mathtt{X}, \mathtt{V}) + \epsilon$
$\mathtt{S} \leftarrow \mathtt{R} \oslash \mathtt{Z}$
$\mathtt{C} \leftarrow BackpropConvInput(\mathtt{V}, \mathtt{S})$
return $\mathtt{X} \odot \mathtt{C}$

Conv() represents a convolutional operation, and *BackpropConvInput()* represents the operation of calculating the gradients of a convolution with respect to the input, as explained in Subsection 2.2.1. $\odot$ and $\oslash$ denote element-wise multiplication and division, as in Algorithms 1 and 2. ϵ is added to all elements of the convolution, in order to avoid dividing by zero in the next step of the algorithm.

Relevance propagation through pooling layers is done in similar fashion. The only difference is replacing the convolutional operation with a pooling operation. This holds true for both max-pooling and average-pooling.

In Algorithm 5, ϵ is added once again to avoid dividing by zero. The Pooling operation in this algorithm can be either max-pooling or average pooling. For this thesis, we decided to use average-pooling for all pooling layers, regardless of the operation in the actual convolutional network. This was done to avoid single high

Algorithm 5 Relevance propagation in pooling layers

Input:

Input activations $\mathtt{X} = \{x_i\}$
Upper-layer relevance vector $\mathtt{R} = \{R_j\}$
Epsilon $\epsilon = 10^{-10}$

Procedure:

$\mathtt{Z} \leftarrow Pool(\mathtt{X}) + \epsilon$
$\mathtt{S} \leftarrow \mathtt{R} \oslash \mathtt{Z}$
$\mathtt{C} \leftarrow BackpropPoolInput(\mathtt{S})$
return $\mathtt{X} \odot \mathtt{C}$

values in the final heatmap. This created more even heatmaps which are easier to interpret, as can be seen in Figure 3.5.

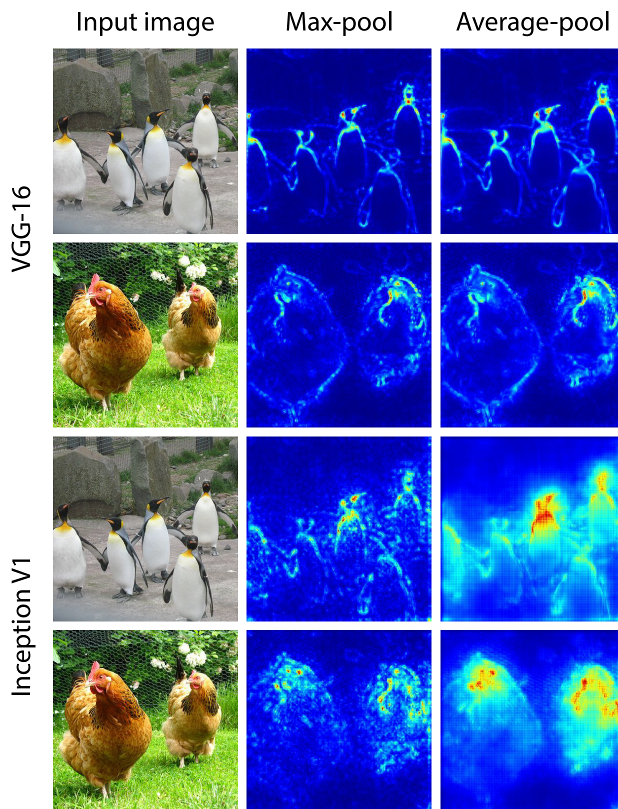


Figure 3.5: Figure displaying heatmaps when using max-pool and when using average pool.

The first two examples in Figure 3.5 are heatmaps generated using the VGG-16 network. The last two examples are generated using the Inception V1 network. It is easier to spot the difference between max and average-pooling in the examples from the Inception network. The reason for this explained in Subsection 4.3.1.

When propagating the relevance from the first convolutional layer to the input image, the $z^{\mathcal{B}}$-Rule expressed in Equation 2.13 is utilized. The values of l_i and h_i might vary from network to network, depending on how the original image is preprocessed before being fed into the network.

In order to generate a Deep Taylor Graph, we started from the output layer and generated one layer corresponding to each layer in the original CNN. This was simplified by the use of Tensorflow, as Tensorflow holds a virtual version of the mathematical graph which we utilized.

3.4.2 Heatmap generation

The final output of the relevance propagation graph will be a three-dimensional matrix of relevance values. The input image consists of three color channels, red, green, and blue. Each of these channels receives their own map of relevances. Layers deeper within the network will have one relevance map for each filter in that layer. In order to create a heatmap, the maps are summed together, before using Algorithm 3 to constrain the values. At this point, one could save the image as grayscale. In order to create a colorful heatmap, we apply a colormap as seen in Figure 3.6, which holds one unique color for each value in the range [0, 255].

Figure 3.6: The colormap used when creating heatmaps.

The colors in Figure 2.7 are ordered from lowest to highest, i.e., from 0 to 255. This method can also be applied to layers which are not the final layer of the relevance propagation. The only difference is the number of heatmaps which are summed together. Examples of such heatmaps can be found in Section 4.3.

3.4.3 Challenges

Inception modules

Propagating the relevance through Inception modules proved to be a challenge, as the modules consist of several convolutional operations which are performed in parallel. The solution was to create a function which split up the Inception module, and propagated through each convolution separately, before combining the resulting relevance values back together.

Generalizing for usage with multiple networks

Implementing Deep Taylor Decomposition for several networks proved to be a bigger task than first anticipated. The main issue was to back-propagate through the network. Different networks have different architectures for their convolutional layers. The Inception network, for instance, utilized batch-normalization(Ioffe and Szegedy, 2015) after each convolution, before applying the ReLU activation function. As mentioned earlier the Inception modules also needed a unique function for propagating the relevance. All in all, it proved to be hard to create a general function which would work for all network architectures without any modifications.

3.5 Combining visualization techniques

In both VGG-16 and Inception V1 there exist hundreds of filters in each convolutional layer. It is possible to visualize each filter, but often the results are very abstract and hard to interpret. The activation of a convolutional layer contains one subset of values for each filter in the layer, as explained in Subsection 2.2.2. When we generate a heatmap of an image, we assign relevance values to each of these subsets. By summing up the relevance values for a subset, we get a relevance score for each group of activation values. As each group is generated by one particular filter, the relevance score displays how much a given filter contributed to the final classification.

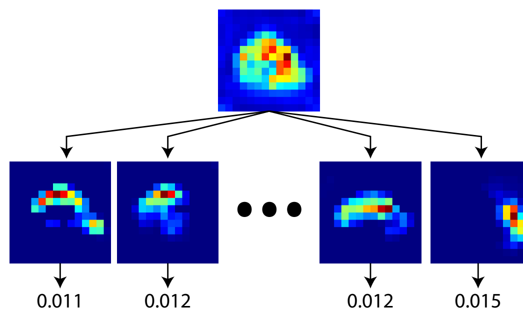


Figure 3.7: Illustration of the process used for calculating relevance scores. The heatmap in this example is generated by a hidden layer in the VGG-16 network.

Figure 3.7 shows how the score for each filter is generated. The set of relevance values is split into one set for each filter. Then the sum of all values within each subset is calculated, which represents the score for the subset. By utilizing this process, we can generate a relevance score for all filters which contribute to a given image. Filters which contribute to the final classification should have some relation to the classification. This could be of use when trying to interpret the result of using

feature-visualization to visualize particular filters, which is something we attempt in Section 4.4.

3.6 Graphical user interface

In order to be able to switch between different functionalities, we created a separate page for each method. At the top of the page, we created a set of tabs, which lets the user switch between the different functionalities of our platform. Since React allows for components to hold a state, one can switch tabs without having to reload previous results upon re-entering a previously used tab. This section shows the different components which exist on each tab and explains their functionalities.

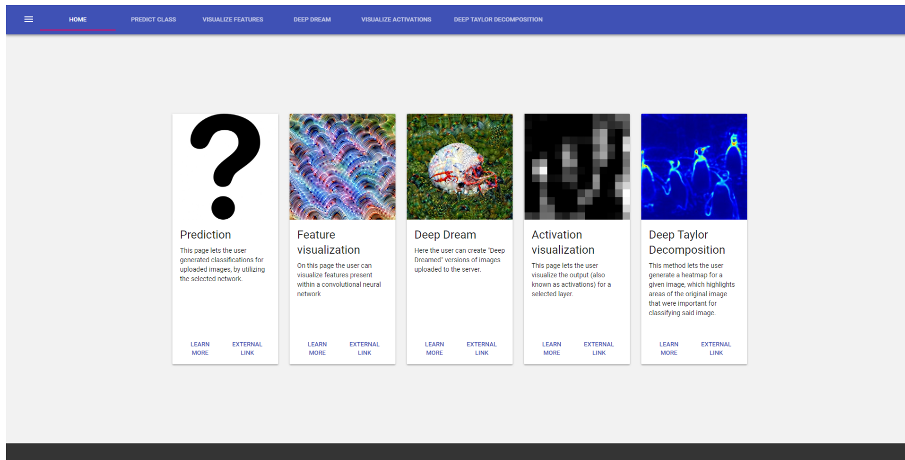
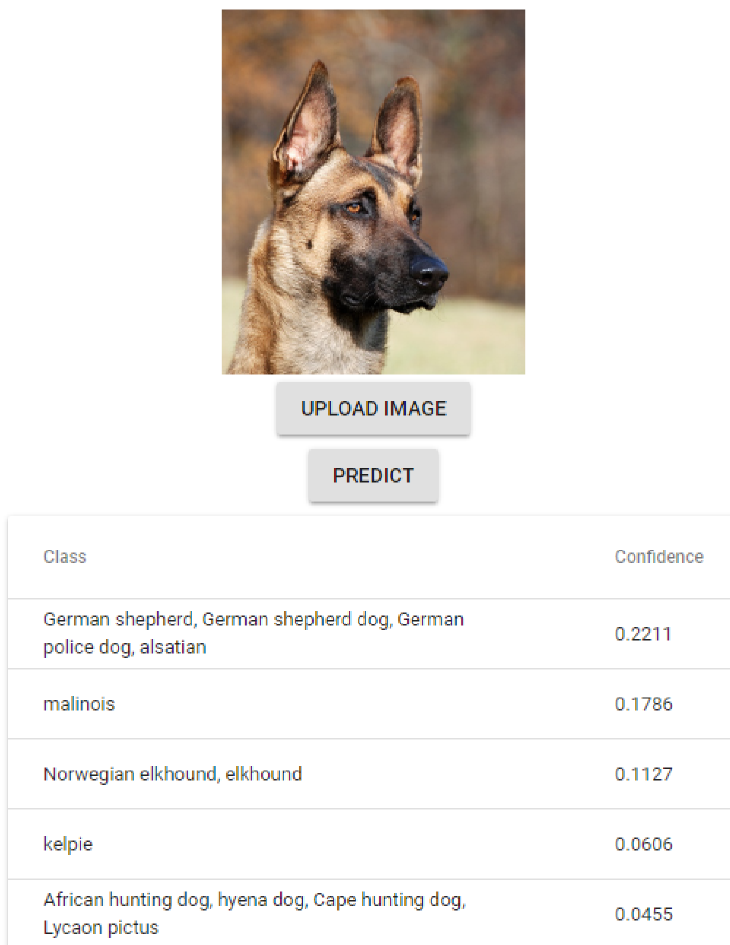


Figure 3.8: Screenshot of the initial page of the user interface.

The page shown in Figure 3.8 holds one information box for each page in the user interface. Each box contains a short explanation of what the page does, as well as two buttons. The leftmost button displays an extended description of the selected page, while the button on the right links to an external source of information on the subject. Each box also holds an image which serves as a preview of the method which the page uses. The user can navigate to the different pages, either by clicking the boxes or by using the tabs in the header f the page. In order to switch between networks, the user can click the button on the left side of the header. This will open a sidebar, where the user can select between different pre-trained networks. Additonal screenshots of each page in the user interface can be found in Appendix C.

3.6.1 Prediction

On the prediction page, the user is shown an image which will be submitted to the CNN for classification. It is also possible to upload a new image using the "Upload image" button. After the classification is computed, the top 5 classes with their corresponding confidence are displayed. It was essential to be able to evaluate any given image, as this is the primary task of the neural network. Knowing the top five classes of an image gives an early indication of what the network is looking for within the image.



Class	Confidence
German shepherd, German shepherd dog, German police dog, alsatian	0.2211
malinois	0.1786
Norwegian elkhound, elkhound	0.1127
kelpie	0.0606
African hunting dog, hyena dog, Cape hunting dog, Lycaon pictus	0.0455

Figure 3.9: Screenshot of the component handling predictions.

3.6.2 Activation visualization

The component for visualizing activations is displayed in Figure 3.10. The user first has to select a layer from the convolutional neural network. When clicking the "Select layer" button, the user is presented with a visual representation of the Network, as shown in Figure 3.12 and 3.13. Here the user can click on any layer in the neural network. Upon selecting a layer, the properties of the selected layer are displayed in the top part of the modal.

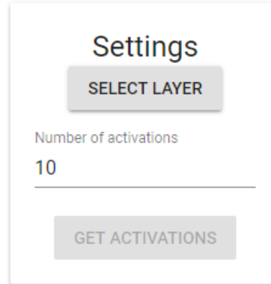


Figure 3.10: Screenshot of the component for activation visualization.

The user can choose how many activations to visualize. Upon pressing "Get Activations," the user interface will request a number of activations chosen by the user. We decided to be able to limit the number of activations, as we would observe that lower ranked activations would be less interesting. This is discussed more in detail in Section 4.2.

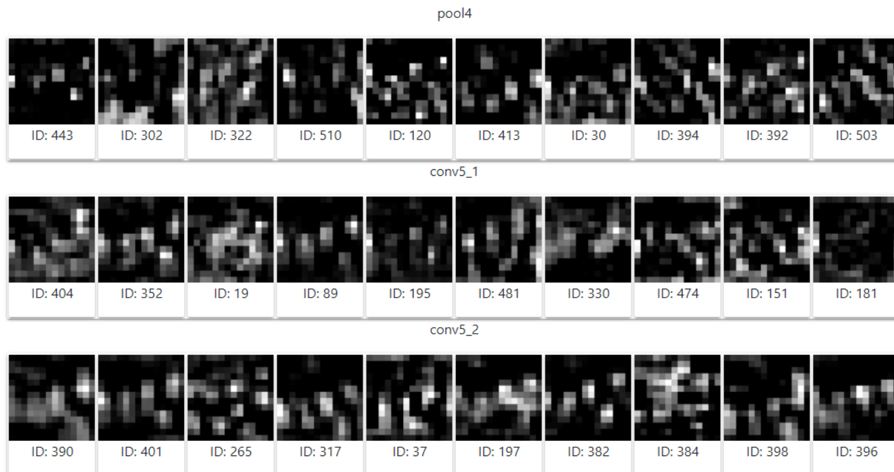


Figure 3.11: Screenshot of the component which displays visualized activations.

The resulting visualizations are displayed as shown in Figure 3.11. Each result has

an ID. The ID corresponds to the index in the output of the selected layer. In this particular example, the top 10 activations from three layers in the VGG-16 network are displayed.

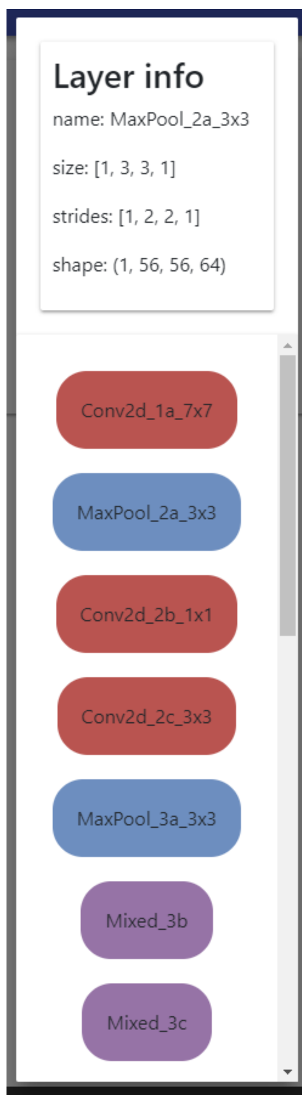


Figure 3.12: Screenshot of graphical representation of the selected network, which in this case is the Inception network.

In Figure 3.12 the different colors correspond to different kinds of layers in neural networks. Red corresponds to convolutional layers, blue is equal to pooling layers, and purple equals Inception modules. The Inception modules can be further expanded, showing the distinct layers within a selected Inception module. At the top

of the visualization rests a box containing information such as stride, output shape and so forth. In Figure 3.12 the selected layer is a pooling layer. The infobox displays the pooling size, stride and output shape of the selected pooling layer. As the selected model has quite a few number of layers, we decided to utilize a scrollbar, in order to fit all layers into one component. The expanded Inception-module in Figure 3.13 is presented the same way as it is in Figure 2.13.

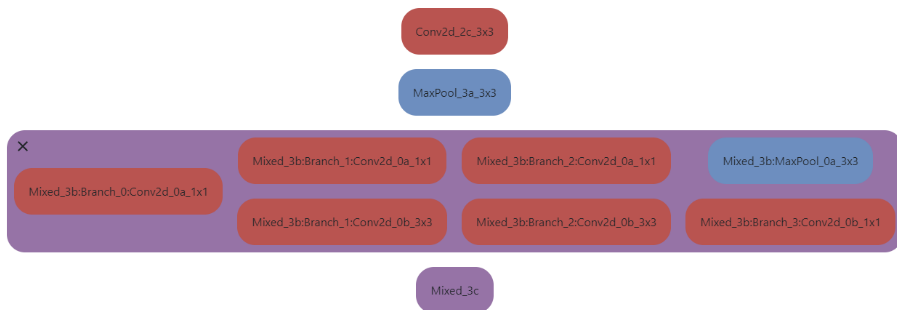


Figure 3.13: Screenshot of an expanded Inception module in the visual representation of the network.

3.6.3 Feature visualization

The component we created for Feature Visualization, is displayed in Figure 3.14. To use this component, one needs to select a layer in the same way as displayed in Subsection 3.6.2. The channel field can be used to select one or more channels to be visualized. If more than one channel is selected and the "visualize" button is pressed, multiple visualizations will be generated, one for each channel. The visualizations will be displayed on the right side of the component. By pressing the "mix" button instead, with multiple channels specified, all channels will be combined into a single loss, and the feature inversion will only produce a single optimized image. If the text field for specifying channels is empty, the entire layer that was chosen will be used for the loss function, as described in Section 3.2.1. There are several different parameters available to alter, which will impact the inversion process in various ways. A list of all available parameters, with short explanations, can be seen in Table 3.1.

In addition to visualizing features by training the visualization network, we also added a "find image examples" button for providing extra clarity regarding the feature in question. Pressing this button will retrieve images that give off the highest activation values with respect to the feature, during prediction as described in Section 3.2.9. By pressing this button, the top 10 image-patches from a dataset of images will be displayed next to the feature visualization itself.

Feature Inversion
SELECT LAYER
Selected layer: Mixed_3b
Channel(s):
11, 34, 5, 248
Input Parameterization:
Naive Fourier Laplacian
Steps: 200 Size: 128 LearningRate: 3 Decorrelate Colors
Padding: 12 Jitter: 8 Rotation: 5 Scale: 0.1
VISUALIZE ! MIX FIND IMAGE EXAMPLES
Visualizing, please wait..
Retrieving examples, please wait..

Figure 3.14: Screenshot of the component for setting parameters during feature inversion. Results appear on the right side of the screen. This component can also be used to mix different channels into the same loss function for a "mixed" feature inversion, or to fetch sample images of the specified objective.

Parameter	**Description**	**Default Value**
Layer Name	The name of the layer we want to visualize. The value is selected through a popup-window	None
Channel(s)	The index of one or more comma-separated channels we want to visualize or mix together. If no channels are specified, the entire layer is visualized	None
Parameterization space	Radiobuttons used to choose between different parameterization spaces	Fourier space
Decorrelate colors	Checkbox used to decorrelate the color channels if checked	Checked
Size	The height and width dimensions of the optimized image	128
Learning-rate	The learning rate used by the optimizer training the network	Naive = 0.2, Fourier = 3.0, Laplacian = 0.05
Steps	Number of steps the optimizer should perform	200
Padding	Number of pixels to pad the parameterization space	12
Jitter	The upper pixel limit that can be used when randomly jittering x and y in both directions	8
Rotation	The upper degree that can be used when randomly rotating the image	5
Scale	The upper and lower percent that the image can be randomly scaled by	0.1

Table 3.1: All the different parameters that can be tuned in the GUI component created for Feature Visualization.

DeepDream

We also created a component for running the DeepDream algorithm over an uploaded image. A screenshot of this component can be seen in Figure 3.15. It is relatively similar to the feature visualization component described above, seeing as it shares a lot of the same parameters. This is caused by the fact that these components share a lot of the same backend functionality when they are used to set up and train their visualization networks. Some of the parameters are removed since they are either irrelevant or not fully compatible with DeepDream in the current implementation.

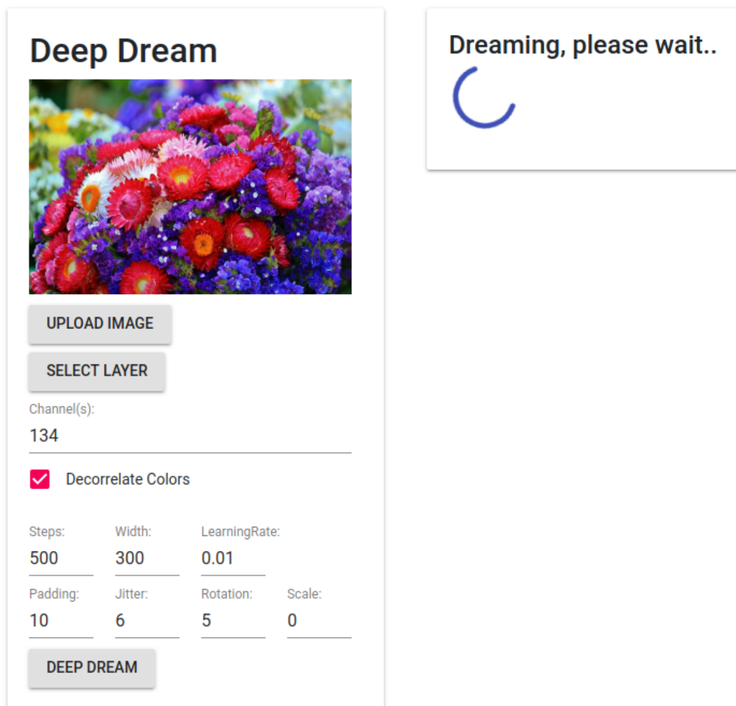


Figure 3.15: Screenshot of the component for setting parameters when creating a "dreamed" image. Results appear on the right. Different channels can be mixed together here as well, in order to create a loss function based on multiple objectives. The uploaded (global state) image, which will be used as input is displayed at the top.

3.6.4 Deep Taylor Decomposition

In order to preview the relevance in the network, we created a component which displays one heatmap for each layer in the convolutional network. Figure 3.16 shows the main component. The user can choose a number of ranked filters, before retrieving the heatmaps. Each Heatmap will be followed by a list of filters, the size equal to what the user chose. The filters will be ranked by their total relevance score. These filters can be used in combination with the technique for feature visualization described in 3.2. This is discussed in detail throughout Section 4.4.

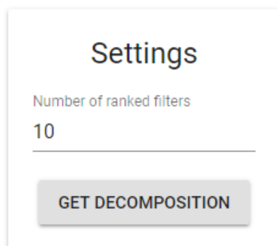


Figure 3.16: Screenshot of the component for generating relevance heatmaps.

The heatmap for each layer is displayed in a separate component, as shown in Figure 3.17. Beneath the heatmap is the list of ranked filters. By clicking the arrow on the right one can either hide or show the list of filters. The button on the left lets the user copy the IDs of all filters in the list to the clipboard. The IDs can be used together with feature visualization, as mentioned earlier. Copying them all independently was perceived to be a tedious task, as one sometimes would like to have up to 100 ranked filters. The score displayed in the list equals the total sum of relevances for the corresponding filter. The ID corresponds to the index of the filter in the weight matrix of the original convolution.

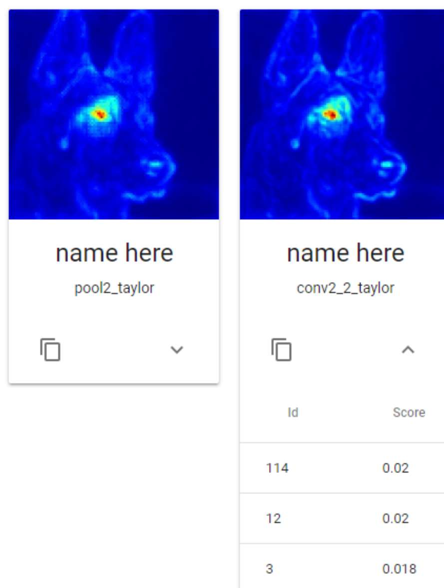


Figure 3.17: Screenshot of the component for displaying relevance heatmaps. In this case two components are displayed, from two different layers in the VGG-16 network.

3.6.5 Challenges

The biggest obstacle when creating the user-interface was the support for multiple neural networks. The graphical layer-selection component, as seen in Figure 3.12 had to be adjusted to fit the different network structures. This included making a custom component for the Inception modules, as seen in Figure 3.13. This was not a huge challenge in itself, but it limits the possibility of creating a general system which works for all network architectures.

3.7 Loading pretrained networks

As previously mentioned, training convolutional networks is a time-consuming process which takes a lot of computing power. Since the methods we implemented focused on visualization, we decided to use pretrained networks for our research. It would have been interesting to visualize throughout the training process, but that was not our point of focus for this thesis. The models used were recreations of the original networks, trained on the 2012 version of the ILSVRC image classification dataset. The two networks we chose to use were the Inception V1 and the VGG-16 network. Both architectures are described in detail in Section 2.3. They achieved

accuracies as seen in Table 3.2. Top-1 accuracy refers to how many percentages of the test set were classified correctly on the first try. Top-5 accuracy refers to how often the correct classification was within the five highest scoring classes.

Network	Top-1 accuracy	Top-5 accuracy
Inception V1	69.8%	89.6%
VGG-16	71.5%	89.8%

Table 3.2: A table displaying the accuracies of the chosen networks.

The models were both retrieved from the Tensorflow Github page[5]. Both of these models were chosen since they arguably are two of the most simple, yet sufficiently deep network architectures, which were available online, and ready to use in combination with Tensorflow. Both mainly consist of two files; one file recreating the architecture in Tensorflow, and one file containing the weights for the pretrained network. In theory, one could easily swap out the weights for another set of weights trained on a different dataset. However, we were not able to find any other pretrained weights of sufficient quality, likely due to the amount of time it takes to train the networks. In order to be able to quickly switch between the two networks in our user interface, we implemented a simple drop-down menu, which lets the user choose which network to use.

As the networks used were recreations of the original architectures, there were some differences. The biggest difference we observed was that the 5x5 convolutions inside each Inception module had been replaced with 3x3 convolutions. In addition, there were some abnormalities within the Inception V1 network, which are discussed further in Chapter 4. When evaluating the results of Deep Taylor Decomposition in Section 4.3 we make use of another version of the Inception V1 network. This version is retrieved from the Tensorflow documentation[6].

[5] https://github.com/tensorflow/models/tree/master/research/slim
[6] https://www.tensorflow.org/versions/r1.0/extend/tool_developers/

Chapter 4

Results and Discussion

In this chapter, we present our findings and evaluate the results in relation to our research questions. We present the result of each visualization method and show examples of how they can be used to analyze and understand a given convolutional neural network. Finally, we evaluate the effect of combining the visualization techniques.

4.1 Feature visualization

When we are teaching a deep neural network to make predictions by training it on a labeled dataset and altering the weights along the way, these weights will over time become more than just their individual values. Patterns emerge and neurons grouped together will after a while start to represent actual concepts of varying complexity. Such encoded concepts appear within hidden layers simply because it is the efficient way of compressing the most important information gathered from the dataset.

In most cases, there are just too many parameters within the hidden layers, and the ways in which they interact with each other to encode information are far too complex for humans to derive something intelligible from. This is what we call the "black box" problem of deep neural networks. Luckily, weights within hidden layers of CNNs automatically structure themselves in such a way that many of the learned concepts can become reasonably comprehensible to humans. We refer to these concepts as features.

Feature inversion, as described throughout Section 3.2 is currently our best approach for depicting features. We apply this method to get a closer understanding of what concepts a certain part of a CNN has learned during training. These are naturally the same concepts the network is looking for when performing predictions

as well. In our implementation, we have utilized several techniques in order to improve the depictions of features. By improving depictions, we attempt to make the fundamental qualities of the feature more comprehensible to humans. This section will go through interesting results from the visualization platform, and look at how the aforementioned techniques impact the quality of generated images of features.

4.1.1 Features in Inception and VGG-16

This section goes through some examples of feature inversions performed on the two sample CNNs, using loss functions based on different types of objectives. As mentioned in Section 3.7, these sample networks are recreated versions of the original Inception and VGG-16 networks, taken from the tf-slim library[1]. Because of small differences in architectures and randomness during training, the weights and features are not the same as in the original networks. Using this platform, we have found a few issues with these recreated networks, which will be elaborated upon throughout the chapter. All results presented under this subsection are generated using the default parameters we regarded as the optimal.3.1

Visualizing layers and channels

In the early layers of most CNNs, there are fewer number of filters used than further into the network. With our visualization graph created in TensorFlow, we are able to visualize every channel from an entire early layer within reasonable time, a couple of minutes at most for <100 channels. These computations were performed using a GPU with 2GB of memory. With more powerful machines running the backend server, simple visualizations could be performed close to real-time.

Figure 4.1 and 4.2 shows visualizations based on entire layers, together with visualizations of every single channel inside the layers, from each respective network. The layers chosen for these examples both appear early in their respective networks, which is why the depicted features appear to be such simple patterns. The next section explores the increasing complexity of features throughout CNNs more in-depth. We can see from these illustrations that only visualizing a layer by itself will give us very limited information about what features the network has learned up to this point. Channels that generally output larger activation values, or have neighboring channels that are visually similar, will naturally appear more prominent in the visualizations of the entire layer. The randomness in the initialization of the parameter space will also have a large impact on the visualization, especially if the loss function is made up of many smaller parts. In other words, singling out specific channels will give us much more useful information than looking at entire layers.

[1] `https://github.com/tensorflow/models/tree/master/research/slim`

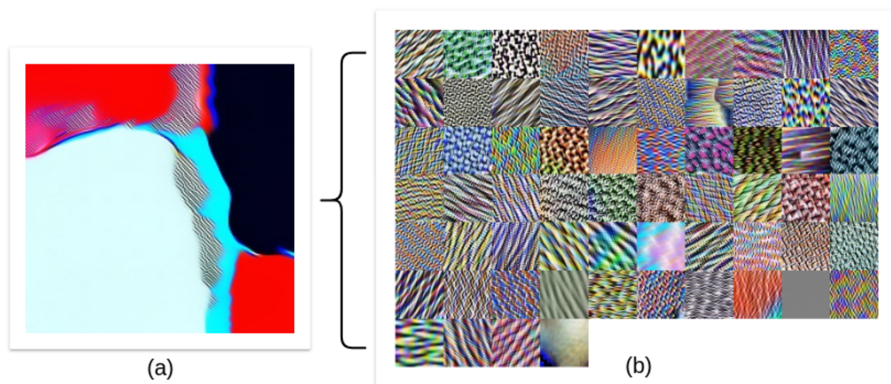


Figure 4.1: Visualization of Conv2d_2b_1x1 from the Inception network. Loss functions are based on the entire layer (a) vs. the 64 individual channels that makes up the layer (b).

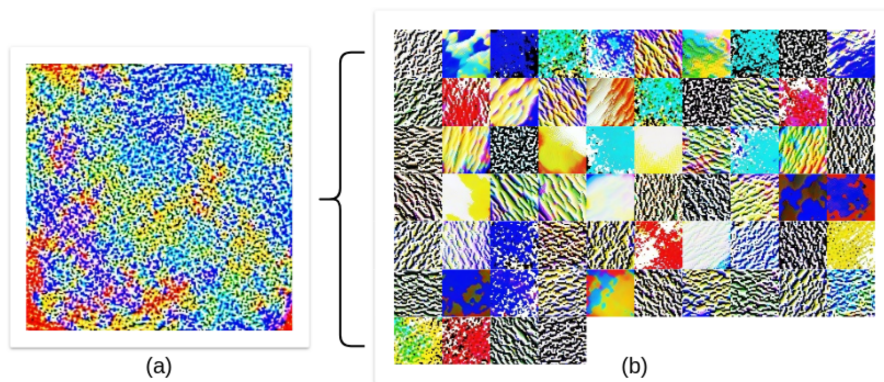


Figure 4.2: Visualization of Conv1_2 from the VGG-16 network. Loss functions are based on the entire layer (a) vs. the 64 individual channels that makes up the layer (b).

If we inspect the visualizations of each separate channel, there might be some unforeseen insights to be gathered. In Figure 4.1 one of the channels located in the bottom right (index: 58) appears entirely grey, meaning it barely could have optimized for the channel at all. Looking at output activations from this specific channel, using the activation visualization tool in our platform as described in Section 3.3, we can inspect it further. Using a lot of different input images, we observed that this channel outputted very small values on all kinds of images, meaning it probably doesn't have a lot of impact on prediction results. We can assume that this channel/neuron is close to being "dead," which means the training process the network has gone through might have room for improvement. Looking at other channel visualizations in the early layers of the networks, we can observe

several features which look similar but are rotated 90 degrees. Utilizing some kind of rotation invariant filters might be a way to improve training time and shrink the number of parameters.

Different levels of abstraction

A neat property of CNNs is the fact that features throughout the network become structured in a hierarchical fashion during training. A feature representing the concept of "trees" is presumably made up of a combination of features from the previous layer, representing concepts like "leaves," "bark," "branches" and so forth. These features are again made up of simpler ones from earlier layers, representing concepts like shapes, colors, basic patterns and so on. The hierarchical structure of CNNs can be demonstrated in practice by visualizing features from layers located at different depths.

In Figure 4.3 and 4.4 we have generated feature inversions of the 5 first channels (index: 0-4) from every convolutional layer throughout VGG-16, located at increasing depth. The last three, fully connected layers are not included. Every example is run for 200 iterations, with the same parameters as defined in Table 3.1. Since CNNs operate the way they do, with filters being run over every channel from the previous layer, there is no special relation between channels with the same index in neighboring layers. The exception is the pooling layers, max pooling in this case, which keeps the number of channels the same, but retains only the most "important" aspects from each channel. The visual impact of the max-pool operation can be observed by comparing the pooling layers in Figure 4.3 and 4.4 to the layers that came directly before.

A property of the VGG-16 network that can be observed is that, as we go further into the network, the features seem to care less and less about which colors they consist of. The visualizations become more colorful as we move closer to the output layer. This is probably an indication that these layers care more about textures and shapes than simple colors at this point. Other CNNs, such as the Inception network keep more color information within the features, even at later layers. The exact causes of these differences are hard to pin down, but the differences in the architectures might be a good starting point. A paper researching the role of colors in deep convolutional networks (Engilberge et al., 2017), also demonstrated that deeper into the network, features become more color invariant.

We can also draw some parallels between the hierarchical structure of CNNs and the way the brain processes visual information. Some aspects of the human visual cortex, which is the closest biological equivalent to CNNs, may become easier to explain through the help of deep neural networks (Khaligh-Razavi and Kriegeskorte, 2014). The human brain is still an incredibly complex system compared to the computer algorithms that are inspired by it, but new connections between the brain and deep convolutional neural networks are still being discovered (Kuzovkin et al., 2018)

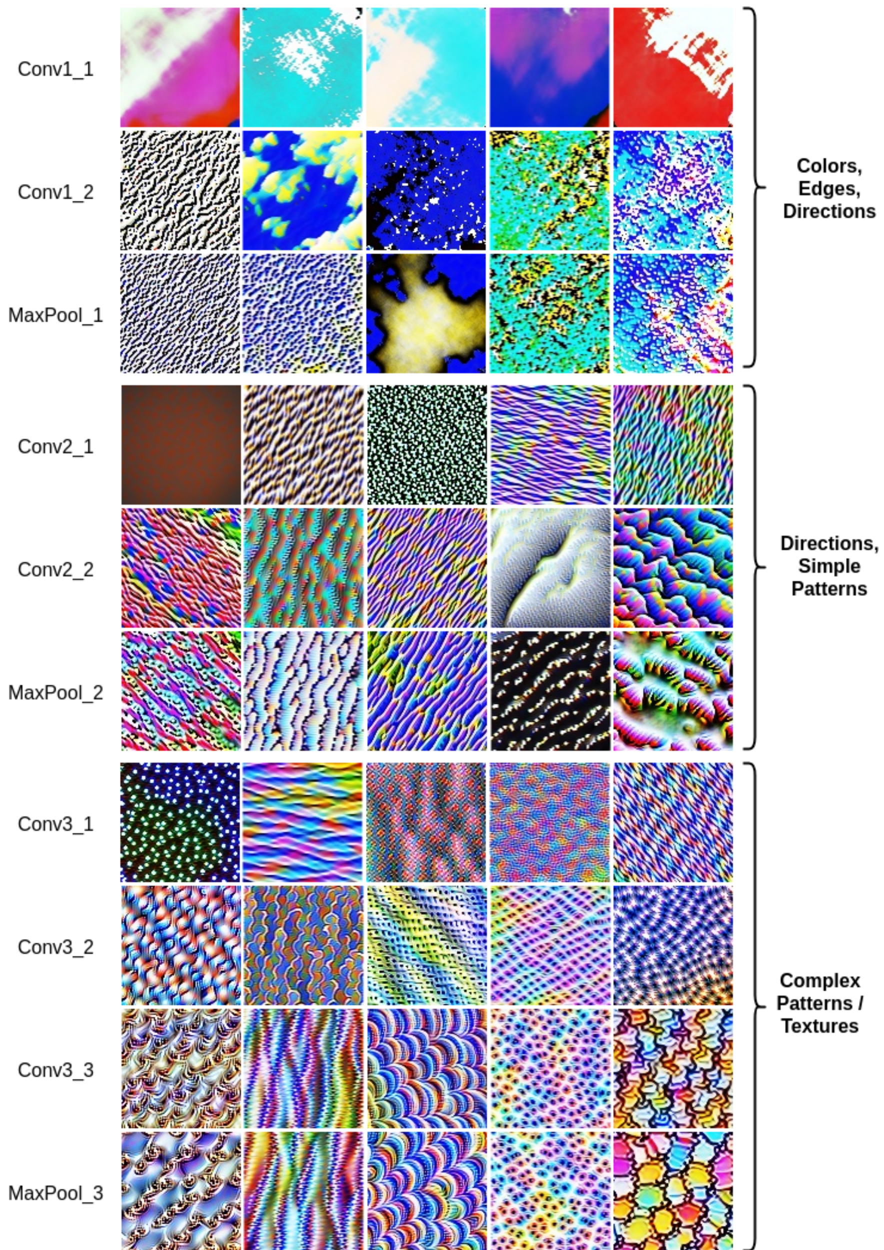


Figure 4.3: Feature visualization of the first 5 channels from the earliest layers (1-10), throughout VGG-16. Features becomes more sophisticated as we go deeper into the network.

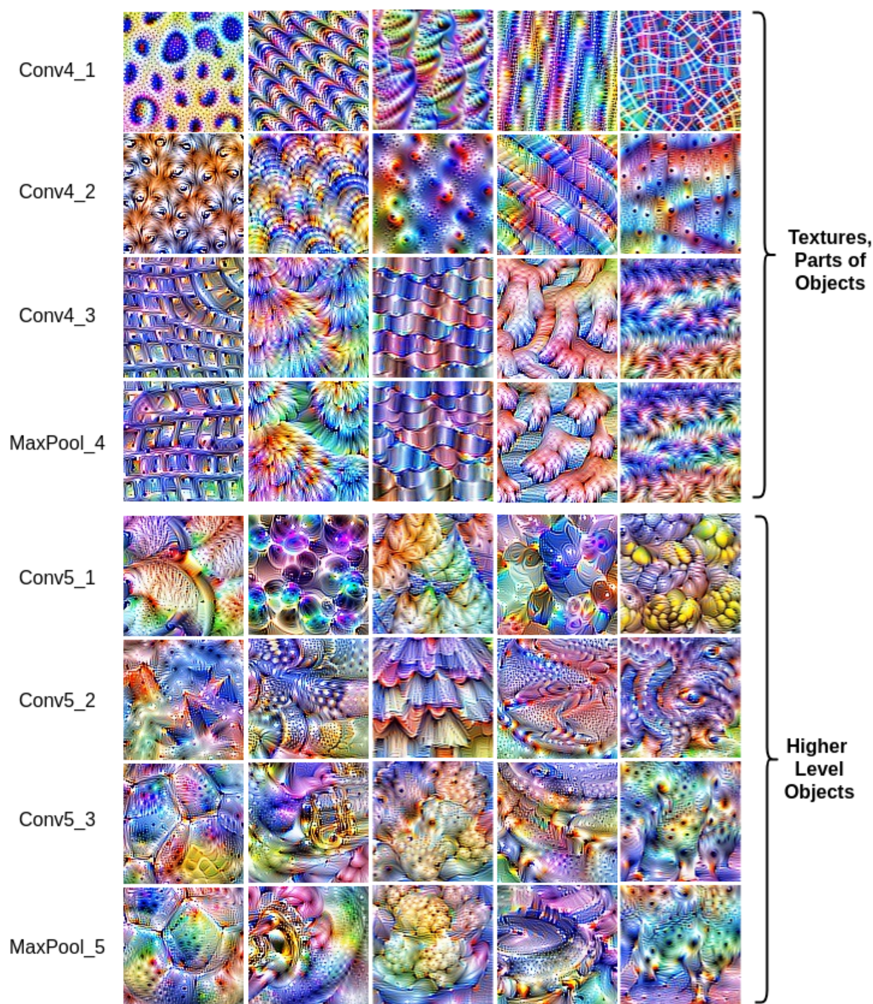


Figure 4.4: Feature visualization of the first 5 channels from layer 11-18 throughout VGG-16. Features becomes more sophisticated as we go deeper into the network.

Visualizing classes

The last layer in a CNN usually represents classes, with one output node for each class. We can visualize these classes exactly the same way we perform visualizations of every other feature inside the network. It is important not to choose the very last layer if it performs a soft-max operation. This is because the soft-max operation suppresses the probabilities of all the other classes, which is not something we want during feature inversion. We select the previous layer instead, which often contains logit values for each class. Using a loss function based on the activations from

one of the neurons in this layer, and we can get a glimpse into what the CNN believes a certain object looks like. By recreating the feature of a class, we might gain some insight into the attributes within the class concept, which plays a role during prediction. In Figure 4.5 we have used our platform to visualize various classes in both the Inception network and in VGG-16. This is one of the few "fair" methods of comparing two CNNs with different architectures against each other, using feature inversion. The last layers have the same dimensions in both networks since they both have been trained on the same set of classes. Looking at features in hidden layers will tell us something about the network itself, but it can be difficult to make reasonable comparisons of two networks with a different number of parameters.

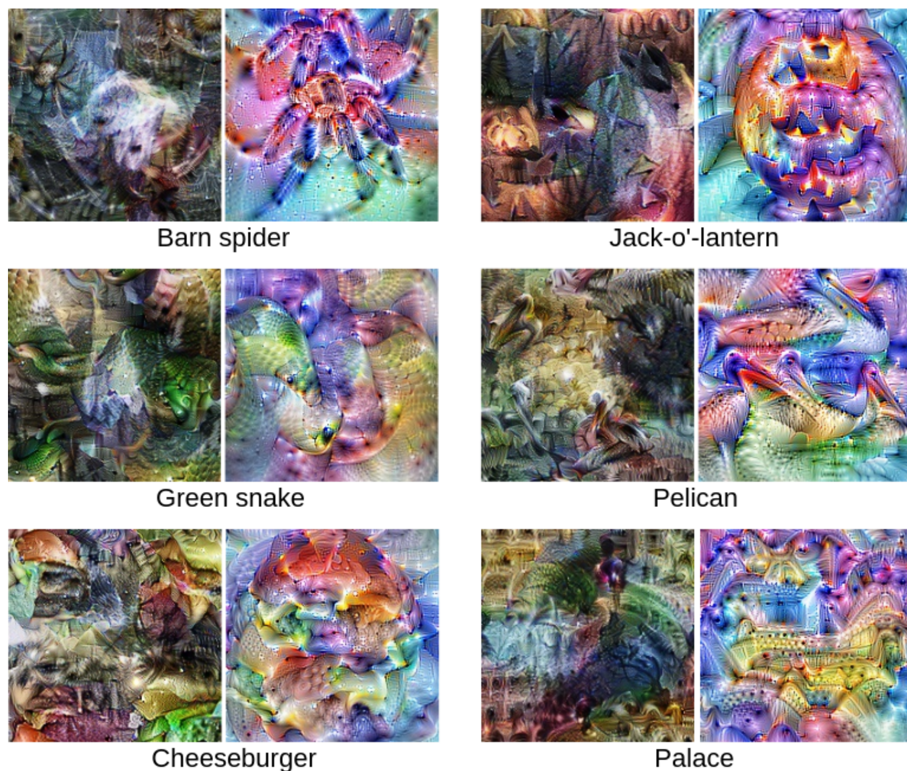


Figure 4.5: Class-visualizations from two different CNNs generated through the platform. Inception on the left and VGG-16 on the right

As we can observe in Figure 4.5, these two networks use vastly different representations of class-objects, even though they both have been trained on exactly the same dataset. The class visualization from the VGG-16 network appears to encompass diverse color schemes, no matter the class. Once again, this might have something to do with the fact that VGG-16 is a much deeper network than Inception, so the layers probably become more color-invariant. Some of these visualizations, such as

VGG-16's version of the barn spider class look a lot like the actual object. Others, like Inception's interpretation of a cheeseburger, appears to be just a mess of different cheeseburger features. The latter is to be expected though, since CNNs that perform image classifications do not concern themselves with the actual structure of objects, only whether the relevant features are present. A dog with multiple heads would still be classified as a dog, to the same degree as a normal dog would.

Figure 4.6: Traffic light class from VGG-16 visualized. A lot of cloud-like features seem to have appeared in the background as well.

Visualizing classes this way can sometimes give us new insights about the different attributes that have become associated with a particular class during training. In Figure 4.6 we have tried to visualize the traffic light class from VGG-16, but ended up with an image containing plenty of clouds in addition to the traffic lights. While it seems the network has been learning the "wrong" attributes of a class, it could be argued that it is advantageous to learn these wrong attributes. If the purpose of the network is to predict the presence of object A in an image, which happens to have a large correlation with the presence of object B, then B should probably be a part of A's concept. This will naturally improve classification of new images, as long as the correlation is still there.

Combining features

As explained in Subsection 3.2.7, we can mix together loss functions based on different parts of the network in any kind of linear fashion. By combining multiple features, we end up discovering even more features. With this in mind, there is an unlimited number of features contained within a trained CNN. The only difficult part is knowing what combinations that might produce interesting results. Figure 4.7 shows examples of multiple features, combined into new ones. Each is based on different channels from the same layer in the inception network, mixed_3b.

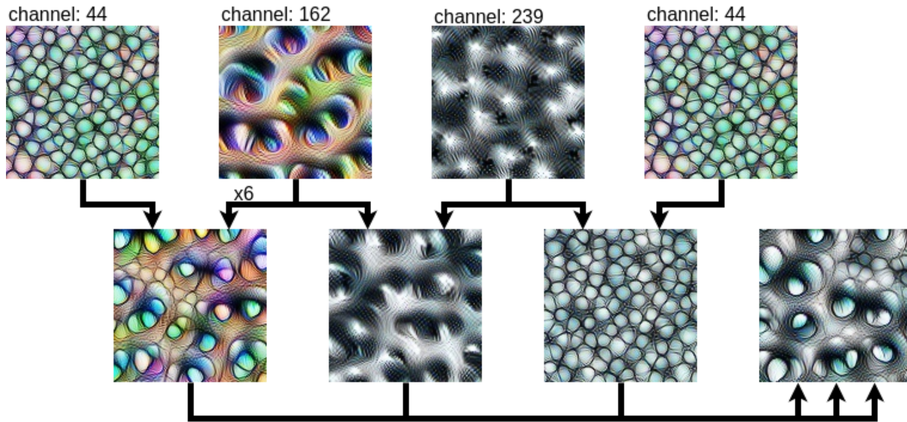


Figure 4.7: Randomly selected features from the layer mixed_3b in the Inception network combined into new features.

When mixing different features together to see how they interact with each other, one might need to give a more significant emphasis to certain elements. In the visualization of channel 44 plus channel 162, as shown in Figure 4.7, we had to multiply the output of channel 162 by a factor of 6 to see them both appear in the combined feature. This is caused by different magnitudes in the filter weights and the complex ways filters might interact with each other during optimization. The factor 6 used in this example was just found through trial and error. Some features encompass more sophisticated concepts than one might expect at first by just taking a look at its visualization. Looking at the feature from channel 239, it is able to make other features appear in black and white. By looking at its activation value when feeding it gray-scale images as well, we can almost certainly say that this is a channel especially trained to recognize black and white images. In addition to better understanding the internals of a CNN, mixing features can also be a great tool for creating computer-generated artworks, especially with high-level features representing more complex objects.

4.1.2 More robust visualizations with stochastic transformations

As explained in Section 3.2.2, we have implemented the option of adding various transformation tensors to the visualization graph, before the input to the pre-trained CNN. These work as regularizers, meant to create more robust feature inversions. Visible improvements, utilizing different transformations can be seen in the results presented below. Every example is created using the same loss function and the same seed when randomly initializing the parameter space. The differences here are only caused by the transformations on the image tensor during optimization, through the transformation graph shown in Figure 3.3. The optimization was

run for 500 iterations with a learning rate of 3, in the Fourier space, with colors decorrelated.

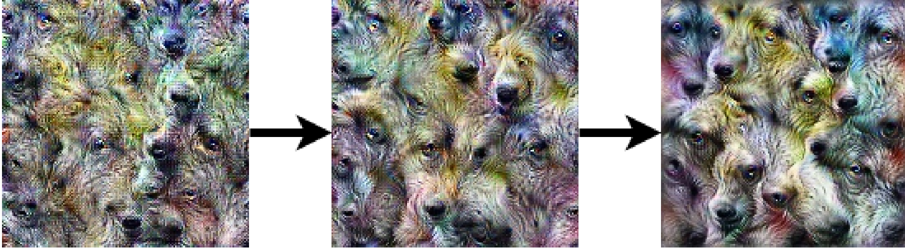


Figure 4.8: Feature inversions from left to right:
jitter = 0, jitter = 1, jitter = 10.

The loss function used for these visualizations is based on (layer: mixed_4c, channel: 191) from the inception network. This feature seems to be depicting dog eyes and snouts among fur. This can be difficult to deduce however, by only looking at the leftmost visualization in Figure 4.8. The image shows the baseline visualization, using no transformations at all. By running the same filters over exactly the same parts of the image multiple times, we get noisy artifacts and saturated pixels throughout the visualization. This can easily be remedied by jittering the image by a random number of pixels. Figure 4.8 shows the "smoothing" effect jittering the image has, which we utilize to generate more natural looking visualizations.

Figure 4.9: Feature inversions from left to right:
angles = 0, angles = ($-5°$, ..., $5°$), angles = ($-180°$, ..., $180°$).

In addition to jittering the image, we can also rotate it by random angles as depicted in Figure 4.9. By slightly rotating the image we are more likely to see additional aspects of the feature while mitigating the chance of getting stuck on a few repeating patterns. This is more of a problem on earlier/simpler layers than the one chosen here though. We need to be careful not to rotate too much, however, as seen in the rightmost image which has been rotated randomly at all angles. While the result might possess some artistic qualities, the cyclic symmetries are not representative of the actual feature.

While still jittering by 10 pixels, and rotating by 5°, we can try to stochastically

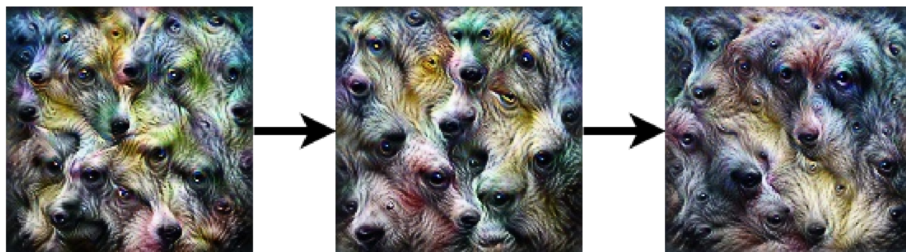


Figure 4.10: Feature inversions from left to right:
scales = 1.0, scales = (0.8, ..., 1.2), scales = (0.1, ..., 1.9).

scale during the optimization as well. Scaling the image will have a clear visual impact as the depicted feature will begin to appear in various sizes. For the same reasons as we were applying rotation, it can help to scale up and down by a small factor like 10%. After extensive testing, it is still a bit unclear whether this transformation is as useful as the jittering and rotation however, in order to make features more comprehensible. Figure 4.10 demonstrates how scaling influences the results, as there appears both smaller and larger dog eyes in the rightmost image.

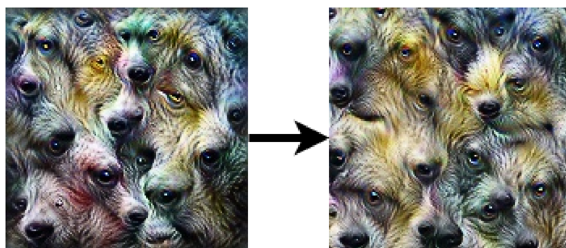


Figure 4.11: Feature inversions from left to right: padding = 0, padding = 12.

In order to avoid the parts close to the edges ending up looking different than the rest of the image, we can add padding. The padding is applied at the beginning as part of the parameter space and is cropped out by another tensor to achieve a more even result as seen in Figure 4.11.

4.1.3 Advantages of alternate parameterization spaces

One of the techniques we applied that seemed to make the largest difference in both speed and quality of visualizations was using a different parameter space. The implementation of these alternative spaces is explained throughout Subsection 3.2.4. The speedup can be easily accounted for by the fact that individual variables in the parameter space now contribute to changes in more than just the color of a single pixel inside the image we optimize. Pixels located spatially close to each other are now more likely to contain similar colors, which is an obvious property

of natural images. This also helps the quality of feature visualizations, making the features easier to recognize.

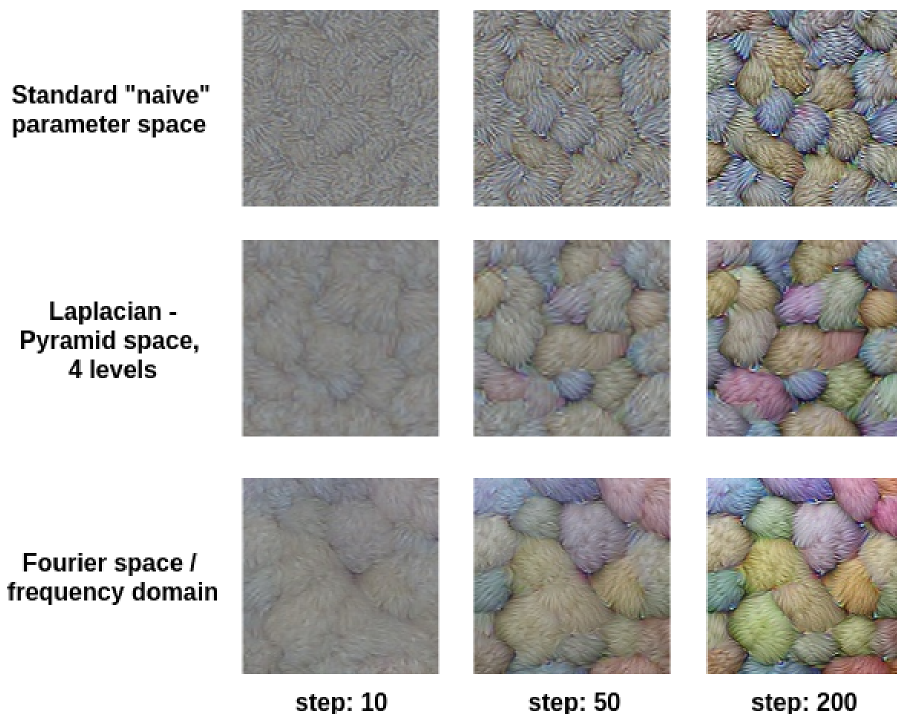


Figure 4.12: Feature visualization at various steps, utilizing different parameterization spaces. Learning-rates are set to: Standard = 0.2, Laplace = 0.05, Fourier = 3.0. The loss function is based on layer: mixed_4c, channel: 134, from the Inception network.

A comparison of three implemented parameter spaces can be seen in Figure 4.12. The feature in this example depicts tufts of hair in various colors. While using a Laplacian Pyramid space is a clear improvement over the standard naive space, it is the frequency space of the Fourier transformation that consistently resulted in the most natural-looking results. It should be noted that it is hard to do a completely fair comparison of convergence rates because of the individual learning rates used in these examples, chosen with respect to the different parameter spaces. The learning rates that were chosen are the ones that gave the most similar, natural looking visualizations after 200 iterations.

Decorrelating colors

Another way we alter the parameter space is by decorrelating the colors, as described in Subsection 3.2.5. When applying decorrelation, we are pointing the

colors in the right directions during optimization by exploiting correlations between colors found in natural images. It is important to keep in mind that we are not making the feature visualizations any more correct, seeing as the layer/channel activation's we try to maximize doesn't increase any by decorrelating. What we instead obtain, are feature visualizations that look more like natural images. This can be observed in Figure 4.13 which shows how the decorrelation alters the optimization. One of the most striking differences we can gather from this example is that decorrelating helps to keep pixel values from getting too saturated, creating too bright, "unnatural" color schemes. To make sure the technique was better than simply suppressing high values, we also compared the decorrelation to using L2-Regularization (Subsection 2.1.6) in the loss-function, suppressing very high or low pixel values within the image.

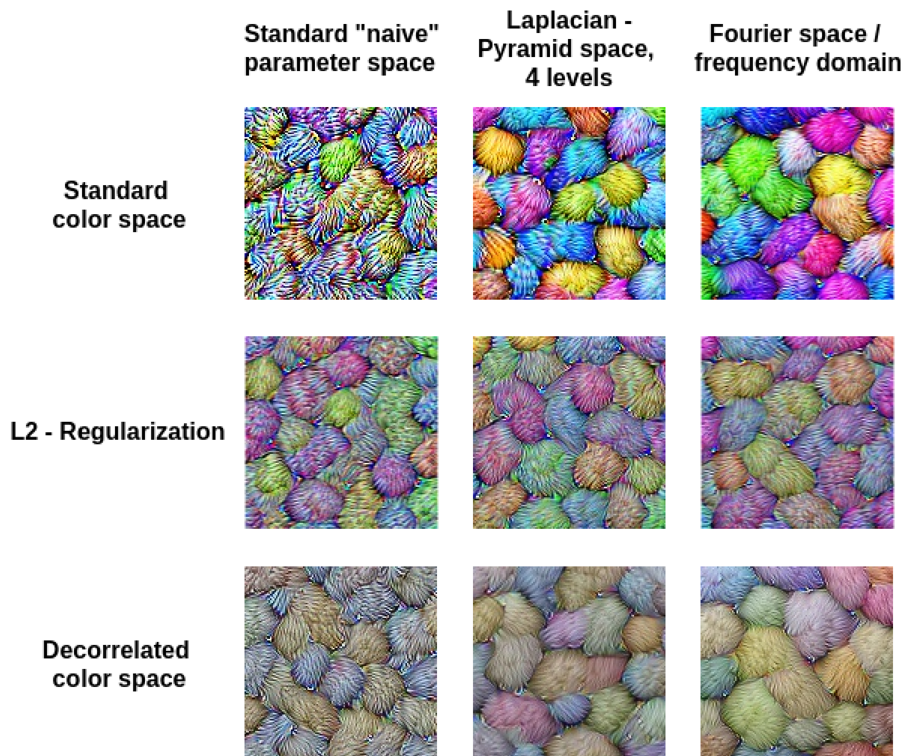


Figure 4.13: Feature inversion after 200 steps, using the same parameters as specified in Figure 4.12. By utilizing decorrelation of colors, as seen in the bottom three visualizations, we get more natural-looking images.

4.1.4 Fetching examples from a dataset of images

In some cases, the optimized image produced through feature inversion might not always be that easy to interpret by itself. This is why we wanted to include a method for fetching images that activated a chosen feature to a large degree as well. By observing the visualization of the feature together with examples of images similar to the feature, the concept encoded into the feature might become easier to interpret. Figure 4.14 shows a few results from this functionality, implemented into the platform as part of the feature visualization component.

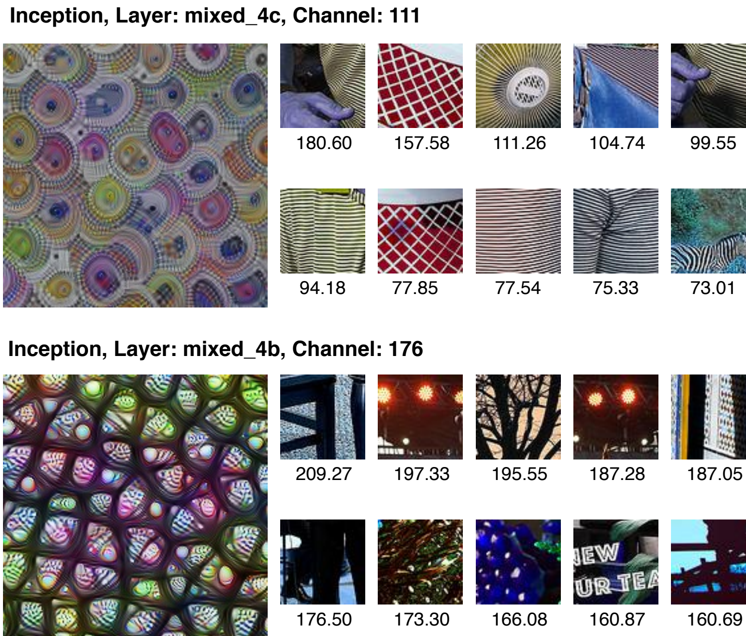


Figure 4.14: Top 10 randomly extracted image-patches taken from the ILSVRC2017 test-set, that had the highest activation values for different channels in the Inception net. The mean output value from the channel is displayed under each image-patch.

The first feature visualization in Figure 4.14, which is based on channel 111, from layer mixed_4c presents a pattern of uneven circles and other strange artifacts. It might be difficult to interpret what kinds of objects or textures this channel reacts positively to in practice. With the method described in Section 3.2.9, we can fetch sample images that activated this feature to a large degree. By looking at the small images to the right of the visualization, we know that the channel reacts positively to curved white stripes.

Unfortunately, the results often turned out a bit poor. The second example shows another feature with corresponding images that almost look random. They all

activated the feature somewhat, but the images will not make it easier to find a semantic description of the feature in question. The "randomness" is caused by the limited number of images and extracted patches used, due to time issues. We extracted and scored 30 patches from all images in the dataset, which took a considerable amount of time. With more powerful GPU's, it would speed up the process, but achieving good enough results in the platform in anything close to real-time is not realistic.

4.1.5 DeepDream

While implementing various techniques in order to improve feature inversion, we realized that these techniques also could be useful in the creation of "dreamed" images, generated through the DeepDream algorithm. Implementing the algorithm into the platform was a fairly simple task, since we could use similar visualization graphs as the ones used in feature inversion. The creation of these dreamed images is more applicable to the field of computer-generated arts than being especially useful in exploring a CNN for scientific reasons. We still wanted to include it into the platform, since there could be unforeseen insights to be gained by feeding actual images into the feature inversion optimization. A few examples of DeepDream run over an example image to create "dreamed" images can be seen in Figure 4.15 below.

These DeepDream examples have been created while making use of the color decorrelation technique in order to achieve more natural colors during optimization. Without altering the colors of the original image too much, we can achieve artistic effects that look a lot like images created with the "Neural Style Transfer" algorithm, described in the paper (Gatys et al., 2015). As explained earlier in Subsection 4.1.1, we miss out on many of the learned features by performing feature inversion using an entire layer, instead of the individual elements that make up the layer. When visualizing a layer through DeepDream instead, more concepts within the layer will usually start to appear. This is because there exist a lot of "features" in the image already, which the algorithm will start enhancing, causing the optimization process to be more diverse in terms of underlying features. This can sometimes reveal biases in the training set. If a lot of dogs or dog-like features suddenly started appearing, this could be an indication that the training set contains a large portion of dog images.

4.1.6 Applications and usefulness

Throughout this section, we have displayed feature inversion results created with the platform, applying different techniques in order to improve the interpretability, while also looking at various types of loss functions. An important question remains as to how useful these techniques would be for actual practical purposes, like debugging a CNN. While the tools we have created related to feature inversion

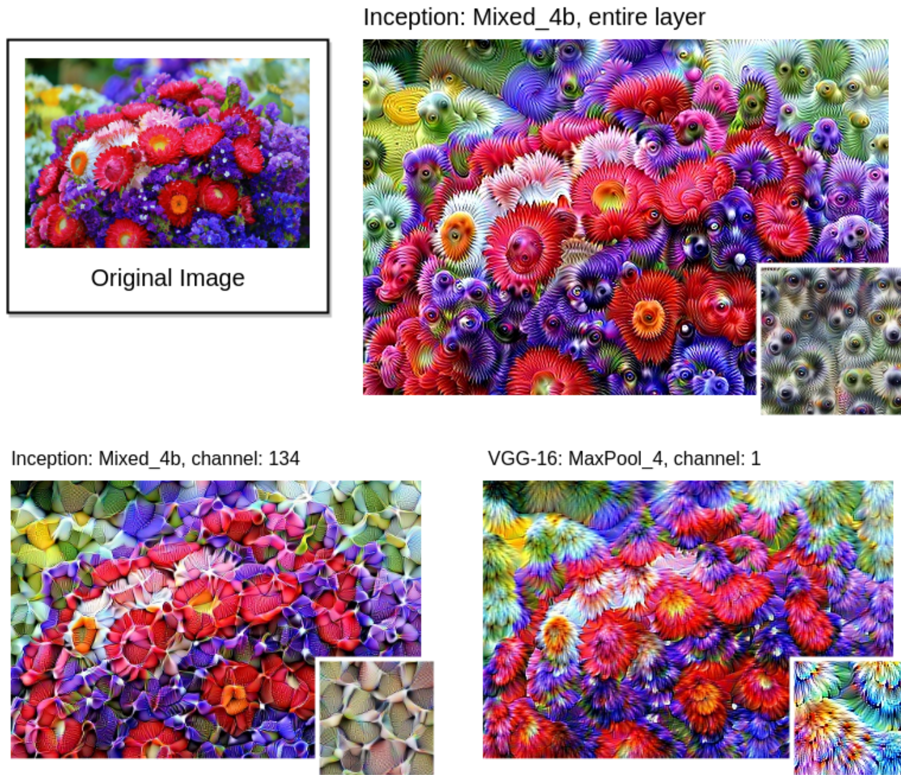


Figure 4.15: Examples of the DeepDream algorithm run over an image of a flower bouquet, using various loss functions from both Inception and VGG-16. The small image in the bottom right of each example shows the corresponding feature visualization using the same loss function.

works great for exploring CNNs, simply for the sake of scientific curiosity, or with the intention of creating computer generated art, it can be harder to find real use cases for these tools by themselves.

One step we could take to make the internals of a pre-trained network more intelligible using the tools above, would be to create a dictionary containing semantic translations of each feature. By visualizing a feature, in addition to fetching sample images that activates it, we could probably give good semantic explanations to most features, although some are quite abstract. Examples of such translations would be phrases like "feathers," "red and blue vertical stripes," "black and white image" and so forth. Depending on the size of the network, this task would probably take a while, but still be feasible in a reasonable amount of time. With such a dictionary in place, we could derive more information about the underlying reasoning behind predictions in real-time. Just as an example, we could know that an image was classified as a traffic light because the "cloud feature" had a high

activation, if a cloud feature had been discovered and given a name.

While performing experiments with the feature inversion tools by themselves, one can stumble over various new insights, discover biases in the training set, redundant filters and other details about the network one might not expect. It might be unreasonable to expect someone to utilize a tool to look for "unforeseen" insights. A better approach would be to perform feature inversions with specific goals in mind. As mentioned in Subsection 4.1.1, we can combine features present in the network in any linear fashion we want to. In other words, we got an incredibly vast feature-space to navigate through when creating a loss function for a new feature. Employing techniques that explore the relationships between channels from different layers, looking at how activation values are propagating throughout the network and other relations, would help in finding more meaningful directions in the feature space. An attempt to do exactly this can be found under Section 4.4.2. Feature inversion by itself may not be all that useful in improving a convolutional network, but combined with other techniques used to navigate the feature space and guide the feature inversion, we believe there is a potential to learn a lot more about "the black box" of CNNs.

4.2 Activation visualization

Visualizing the activations of each layer in a convolutional network gives an impression of how the output changes from layer to layer. As we can see in Figure 4.16 the output goes from being similar to the input image to only having high activation values in certain regions. The final activations in the VGG-16 network have high activations in regions which correspond to the head of the parrot in the original image. In the Inception network whose activations can be seen in Figure A.1, the region with the highest activation is more evenly adjusted over the entire bird.

This technique can also be used to identify dead neurons. In the context of neural networks the term "dead neuron" refers to when a neuron outputs the same value, regardless of input, and therefore the gradient of the neuron becomes zero. When this happens, the neuron is not contributing to classifying the input image. Dead neurons often occur in combination with the ReLU-activation function, as a negative input always yields a zero value. Many dead neurons might be an indication of using the wrong parameters during the training process, such as a high learning-rate. When looking at the visualized activations, images which are all black, i.e., have no activation values above zero, indicate a dead set of neurons, referred to as a dead-filter. It would not be sufficient to make assumptions from looking at the activations of one single input image, as the filter might respond to features which are not present in that image. A possible drawback of this technique is its scalability with the increasing depth of a neural network. When increasing the amount of filters in a convolutional layer, it becomes increasingly difficult to interpret the resulting activations. Certain filters might only react to features in

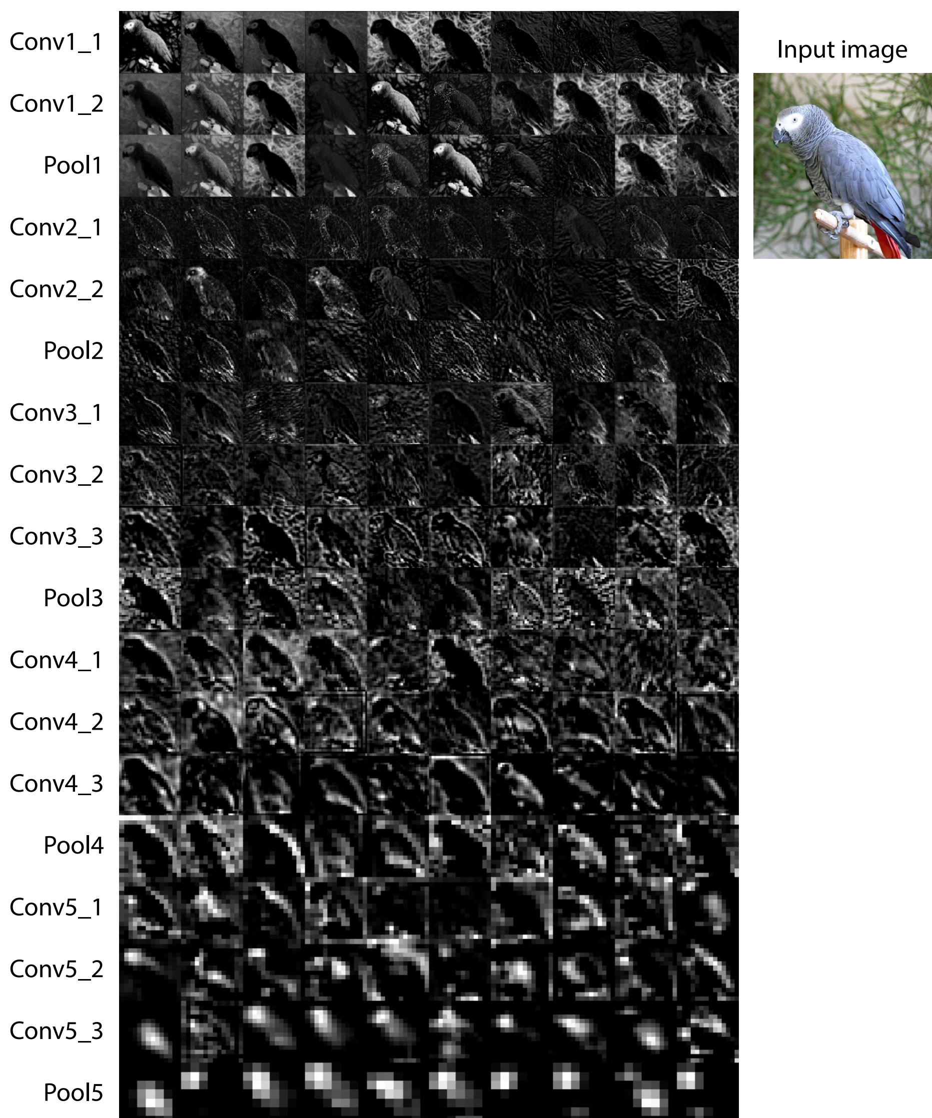


Figure 4.16: Example showing the top 10 activations from all layers. The input image shows an African grey parrot, and was classified correctly as such. The activations are taken from the VGG-16 network.

one particular class, which might result in them appearing as "dead-filters." Additionally, it might be difficult to distinguish which feature a filter reacts to, as there might be several different features within an image. Despite these drawbacks, Activation Visualization still appears as a solid technique for exploring convolutional

networks.

4.3 Deep Taylor Decomposition

By utilizing Deep Taylor Decomposition, we were able to get a better impression of which parts of an image were responsible for the resulting classification. Additionally, we observed a certain arrangement of relevance values, often referred to as checkerboard artifacts. We were also able to find some abnormalities in the pre-trained version of the Inception V1 network.

Figure 4.17 shows several heatmaps generated using Deep Taylor decomposition. All input images were correctly classified by both networks. The heatmaps generated by the VGG-16 network appear to be clearer and more readable than those generated by the Inception network. The heatmaps from the Inception network seem to have a certain pattern to them, which is explained in Subsection 4.3.1. When looking at the heatmaps originating from the first image, one can see a region of high values residing around the eye of the dog, as well as the nostrils. This would suggest that the network has learned the concept of eyes and nostrils, and recognizes it in new, previously unseen images.

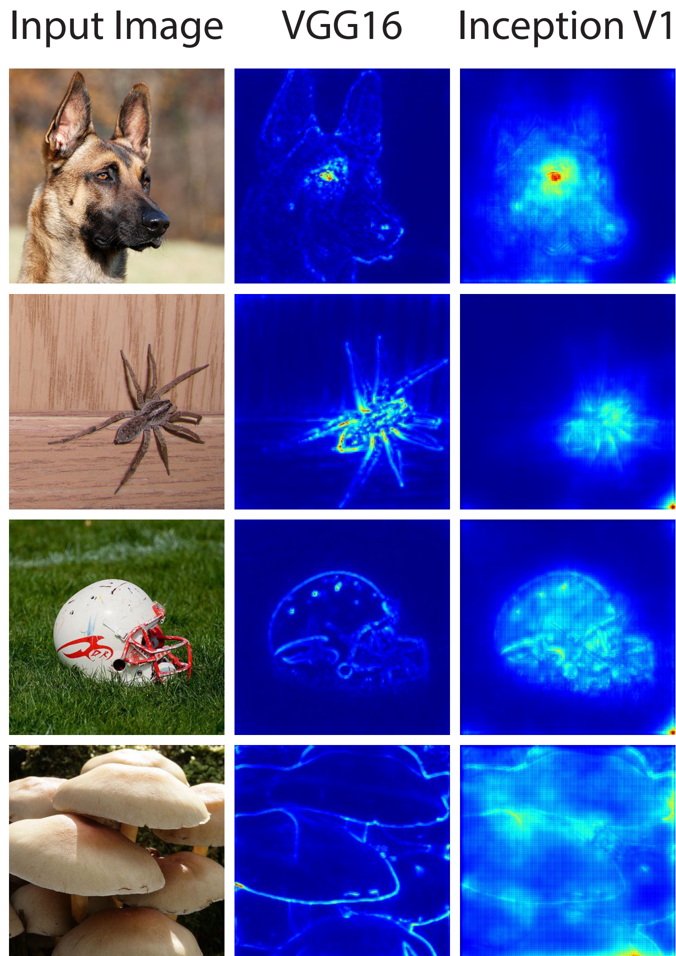


Figure 4.17: Example of heatmaps generated with Deep Taylor Decomposition.

4.3.1 Checkerboard artifacts

The pattern seen in the heatmaps of the Inception V1 network is often referred to as checkerboard artifacts. This can be explained by looking at the structure of the Inception network. The first three Max-pool layers use a pooling region of [3,3], with a stride of 2. When propagating the relevance certain output values will be part of several pooling regions, meaning they receive relevance more often than other output values, as seen in Figure 4.18.

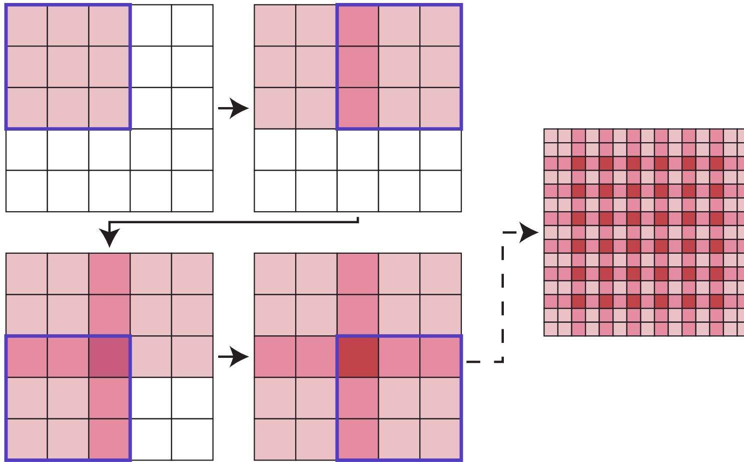


Figure 4.18: Figure displaying how checkerboard artifacts occur.

Figure 4.18 shows the operation of propagating relevance through a pooling layer. The original input is of size [2,2]. The pooling operation uses a pooling region of [3,3] with a stride of 2. as one can observe, the center pixel is part of all four pooling regions, giving it relevance from all four input values. When doing the same operation with an input of bigger size, one will get the pattern shown in the last part of the figure.

Checkerboard artifacts have also been observed in other visualization techniques, such as Deconvolutional Networks(Odena et al., 2016). The reason we experience them can be traced back to an adjustment we made in our implementation. As mentioned in Subsection 3.4.1 we chose to propagate relevance through max-pooling layers by treating them as if the were average-pooling layers. When looking at Figure 3.5 we can see that this choice greatly influenced the heatmaps generated by the Inception network. However, we still argue that the heatmaps generated with average-pooling are more readable. An example of this is the first set of penguin heatmaps from Figure 3.5. The heatmap generated with max-pool shows the outline of the penguins, but it is hard to interpret why it received its label. Also, there is a lot of noise between the different penguins. In the second heatmap created utilizing average-pooling, it is easier to spot that the center penguin holds most of the relevance. Especially the neck and head of the penguin received a lot of

relevance, as these are distinct features of this particular species of penguin.

4.3.2 Abnormalities in the Inception Network

When looking at the heatmaps generated from the Inception network, one can observe that there often is a small region with high values in the corners. This seems odd, as the corners do not seem to contain any valuable information. There is no clear link between the actual classification and the region of high values in the corner. The high activations in seemingly irrelevant parts of the image make it harder to interpret which part of the image are actually relevant for the classification.

In order to confirm that the issue was related to the particular network used, we generated alternative heatmaps using another version of the Inception V1 network. The basic network structure is the same in both networks. However, the alternative version of the Inception network uses the correct 5x5 convolutions, instead of 3x3 convolutions as mentioned earlier.

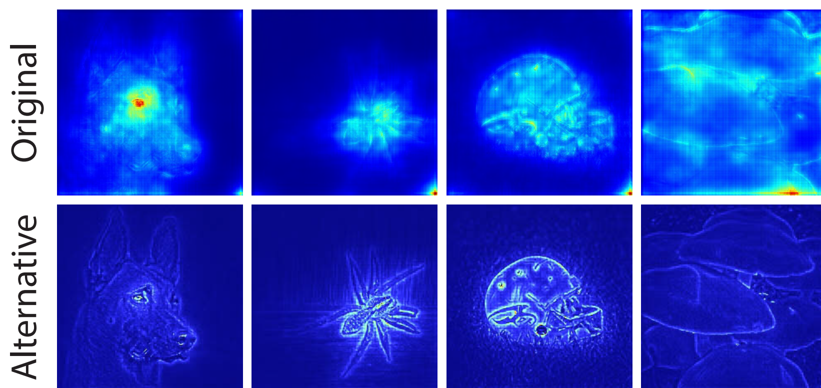


Figure 4.19: Comparison of heatmaps from both versions of Inception V1.

Figure 4.19 shows the heatmaps generated by both versions of the Inception V1 network. Original refers to the original network as seen in Figure 4.17, while alternative refers to the other version of the Inception network, which uses the correct 5x5 convolutions. The alternative heatmaps show a considerable improvement in terms of readability. The heatmap values are more concentrated, and there is no trace of high values in the corner regions of each heatmap. The difference in quality between the generated heatmaps is most likely a result of using 3x3 instead of 5x5 convolutions. There is no real benefit in having a second 3x3 convolutional operations in each inception module, instead of increasing the number of filters in the first one. Another possibly relevant factor is differences in the training process. There is no information accessible on how the networks are trained and which parameters were used, such as batch-size and number of epochs.

4.3.3 Recognizing general features using heatmaps

The generated heatmaps seem to highlight parts of the image which are vital in order to classify it correctly. If a feature in an image is vital for its classification, it should be highlighted in other images from the same class. In order to verify this statement, we looked at the heatmap of eight different images from the class African grey parrot.

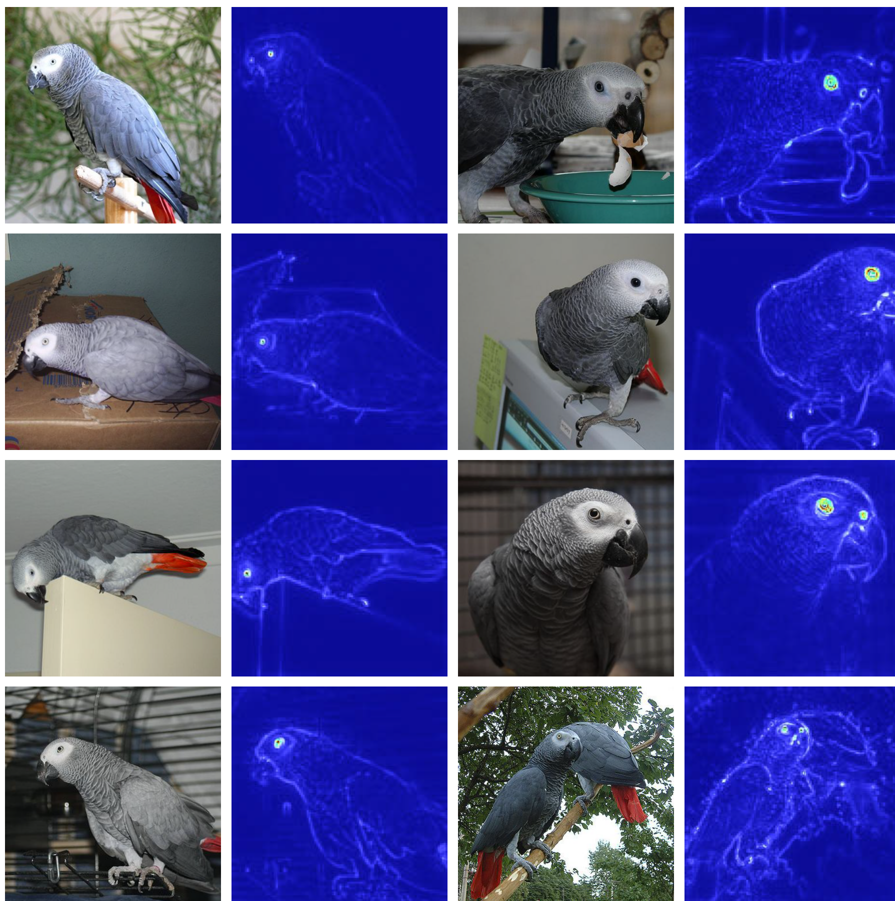


Figure 4.20: Images and resulting heatmaps. Each image is classified as an African grey parrot.

All heatmaps in Figure 4.20 are generated using the VGG-16 network. A distinct feature which all heatmaps share is highlighting the eye of the parrot. The nostrils of the parrots also seem to be highlighted in images where they are clearly visible. There is no significant difference in confidence for images with and without highlighted nostrils. All images are classified as African grey parrot with a confidence

between 99 and 100%. Another often highlighted feature is the edge between the beak and the head. From these examples, it seems pretty clear that the network has learned to recognize different features which combine to form the concept of African grey parrot.

4.3.4 Relevance through all layers

In order to better understand the propagation of the relevance throughout the layers of a CNN, we generated one heatmap for each layer in the network. These heatmaps show the propagation of the total relevance from the first to the last convolutional layer, in reverse order, i.e., the order in which the relevance is propagated through them.

As seen in Figure 4.21 the first heatmap is an indistinguishable collection of heatmap values, with no clear patterns. As the dimensions of the heatmap increase, a contour of objects appears. It does not take long before the highest heatmap values are distributed on the penguins in the original image. As we come closer to the input layer of the original network, more details appear in the generated heatmaps. It becomes apparent that the head of the penguin, along with its unique color around the neck area are features which are relevant in order to classify the image as a penguin.

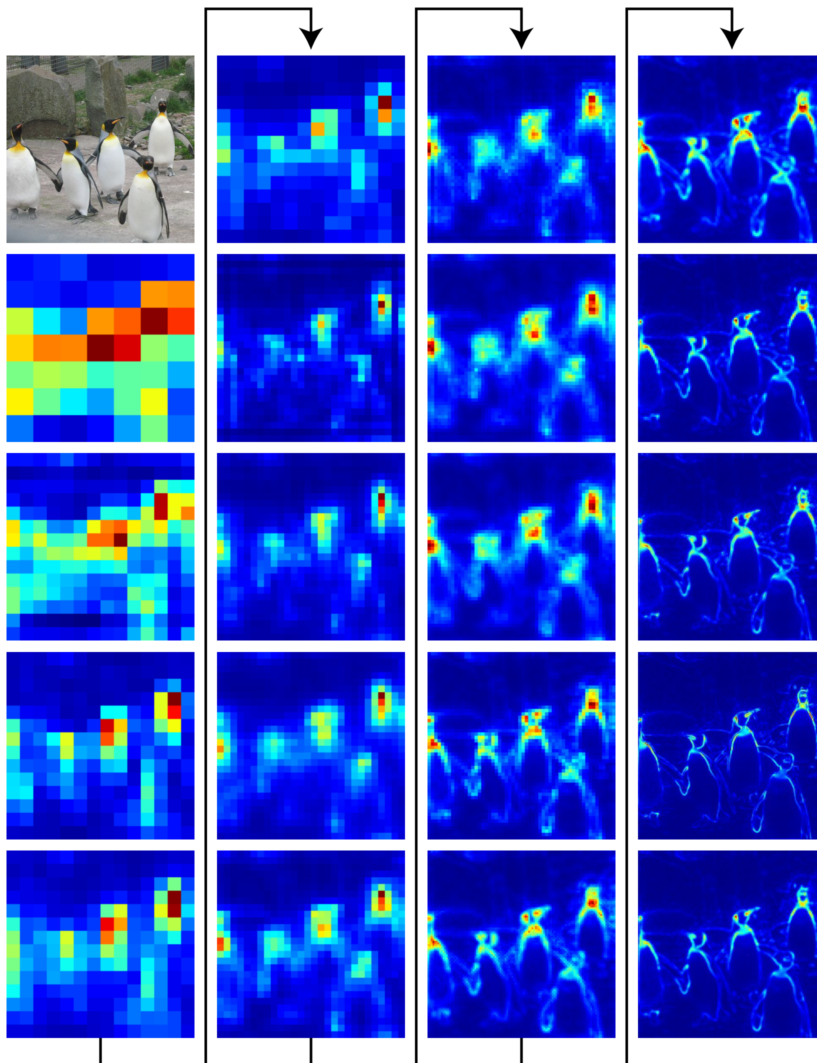


Figure 4.21: Heatmaps showing the relevance layer-by-layer. The first image is the original input image. The relevance starts at the first convolutional layer closest to the output layer, and ends at the input-layer of the network.

4.3.5 Generating heatmaps for wrongly classified images

In order to evaluate Deep Taylor Decomposition further, we decided to look at the heatmaps of images which received the wrong classification. We utilized a labeling interface[2] for the ILSVRC, in order to retrieve 300 images for which the network would fail to generate the same classification as the Imagenet database. The set of images can be roughly divided into four groups.

- **Similar class -** A few of the wrongly classified images received a label which was closely related to the correct class. For instance, the network would classify an image as one type of dog, while the correct class was another type of dog.
- **Multiple objects -** Some images contained several objects. In these images it was hard to identify which object to classify. Even for humans, it was sometimes not trivial which object yielded the correct classification for the image.
- **Wrong label -** Another group of wrongly classified images were those with faulty labels in the Imagenet database. For some images, the Imagenet classification was outright wrong. Other times the image contained an object which looked similar to the Imagenet classification but still was mislabeled.
- **Other -** The last group contained images where there was no trivial correlation between the image and the classification generated by the network.

Figure 4.22 shows one example for each class mentioned above. Table 4.1 shows the Imagenet label for each image, along with the label generated by the neural network. For these examples, we used the VGG-16 network to classify the images, and generate heatmaps.

id	Imagenet classification	Generated classification
1	Common Iguana	Green Lizard
2	Airship	Traffick light
3	Tick	black and gold garden spider
4	Toilet Paper	washbasin, washbowl

Table 4.1: List of classifications for the images in Figure 4.22. The Imagenet classification is what the image is labeled as in the Imagenet database. Generated classification is the class generated by the network.

The first image received a classification which is similar to the Imagenet class. The relationship between the Imagenet classification and the generated label is understandable, as they both are animals from the same species. The second image contains both the Imagenet class and the generated classification. However the view of the airship is obstructed, and therefore the traffic-light is assumed as the most relevant object within the image. This assumption is supported by the heatmap

[2] https://cs.stanford.edu/people/karpathy/ilsvrc/

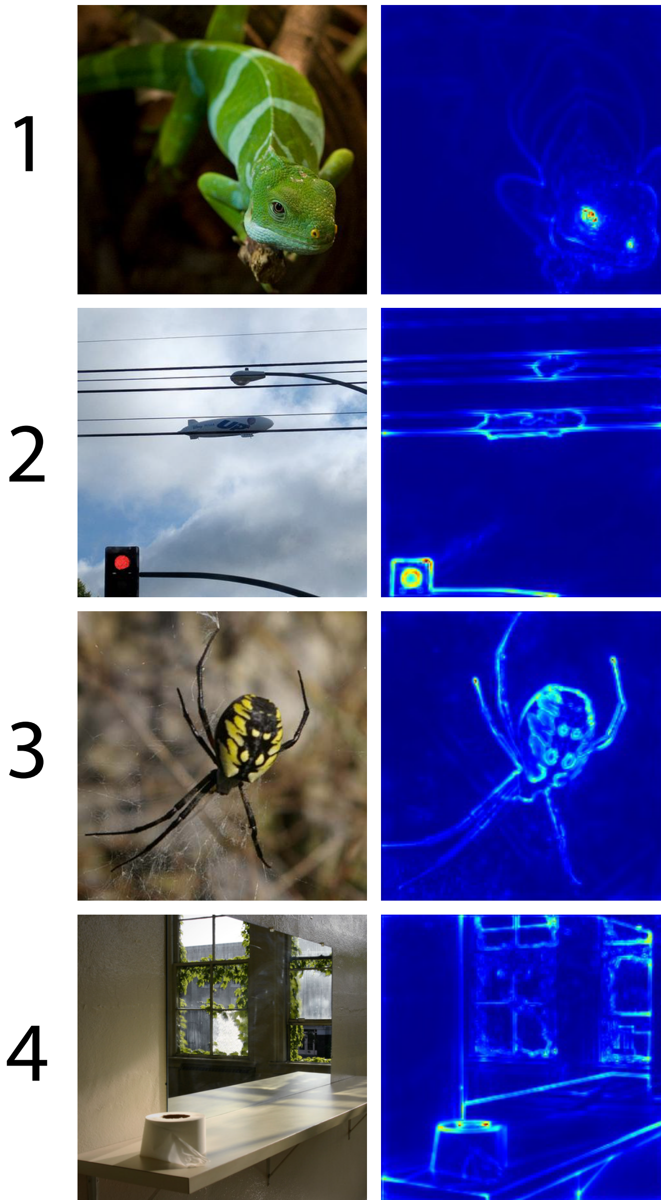


Figure 4.22: Examples of images which were wrongly classified, along with their corresponding heatmaps.

of the image, where one can see that the traffic-light receives a lot of high values. In the third image the original image is classified wrongly as a tick. However, the network seems to have learned the correct classification. Additionally, the

heatmap shows a high relevance score for features of the spider, which supports the assumption that the network has learned to classify this class correctly. The final image is classified as *Toilet Paper* in the Imagenet Database. However the VGG-16 network generates the classification *Washbasin, Washbowl*, and there seems to be no elementary reason why. The heatmap of the generated image does not offer any direct information as to why the image was wrongly classified. A possible explanation is that the image contains certain objects or features which also exist in images containing a washbasin, such as the mirror and the general color-scheme of the image. However, there is no further proof to support this theory.

Most of the wrongly classified images belong within the three first groups of images. A significant percentage of the images were either wrongly classified, or contained several different objects, making it hard to determine which was the true class of the image. There were also a lot of images which received closely related classes, especially when the object was an animal. This can be presumed to be a result of animals sharing several features, such as eyes and limbs.

4.3.6 Applications and usefulness

Creating heatmaps by utilizing Deep Taylor Decomposition shows promising results when it comes to identifying features which a network has learned in order to distinguish between different classes. Visualizing the relevance layer-by-layer offers an understanding of how the relevance is distributed to create the final heatmap. Often one can get an initial assumption on why an image received a wrong classification. Generating a heatmap for such an image can reinforce the initial assumption, by establishing which parts of the image were important for the generated classification. However, for certain images the heatmap does not offer a lot of additional information.

4.4 Combining visualization techniques

This section displays the results from attempting to combine different visualization techniques, in order to explain the results of feature visualization. The results from visualizing different filters in Section 4.1 were intriguing but sometimes hard to interpret. We attempted to use relevance propagation as guidance for feature visualization, in order to select which filters we wanted to visualize.

4.4.1 Visualizing individual filters

First, we calculated relevance scores for each filter in each layer, using the method explained in Section 3.5. After calculating all scores, we visualized the filters in each layer which had the highest relevance values. An example of this can be seen in Appendix B. The class of the original image was *King Penguin*. One can argue

that the visualizations of layer *Conv5_2* show some resemblance to features of a penguin, such as the eyes and beaks, but not enough to draw any conclusions. In general, utilizing relevance scores to choose which filters to visualize showed close to no improvement towards interpreting the generated visualizations. The visualizations still consist of seemingly random patterns, with no straightforward relation to the classification of the image used for relevance propagation. It seems like the lower layers, which hold more abstract concepts, to some degree resemble features present in the original image, but there is not enough evidence present to support this claim. We believe one reason for this is the fact that each filter has a relatively low relevance score, ranging from 1-3% of the total relevance. It seems intuitive that a combination of different filters is needed to generate a classification.

4.4.2 Combining relevant filters

Figure 4.23 strengthens the theory mentioned in the previous subsection. The graph shows the sum of relevance for all filters within one layer, starting from highest ranked to lowest ranked filter. Even the filters with the highest relevance values only account for a fraction of the total relevance within the layer. The accumulated relevance for the first 100 filters is approximately 50% of the total relevance in Figure 4.23. In total there are 512 different filters in the layer conv5_2, and only a small portion have received a relevance score of 0. Instead of looking at the visualization of single filters, a better approach would be to combine the visualization of several filters together to produce a single visualization.

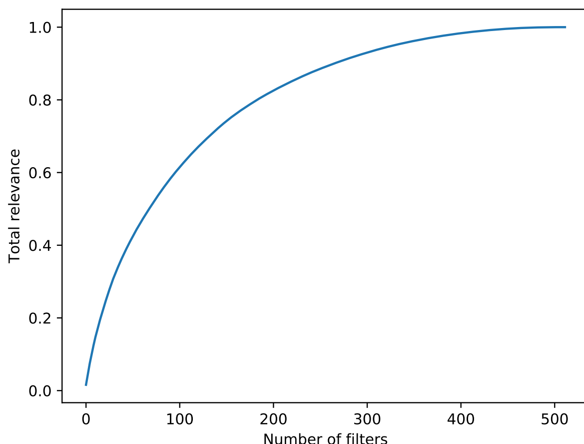


Figure 4.23: Graph showing the accumulated relevance for layer Conv5_2 in the VGG-16 network.

In Figure 4.24 one can see an example of optimizing for several filters at once. The early layers are still not easy to explain and show seemingly random patterns. However, as we reach the later layers of the network, the patterns start to show some resemblance to the class king penguin, such as beaks, eyes and the general shape of king penguins. In general, the relation to the original class seems more clear compared to the results of optimizing for single filters. However, the visualizations are still open to interpretation, and it can be challenging to identify features within the generated visualizations.

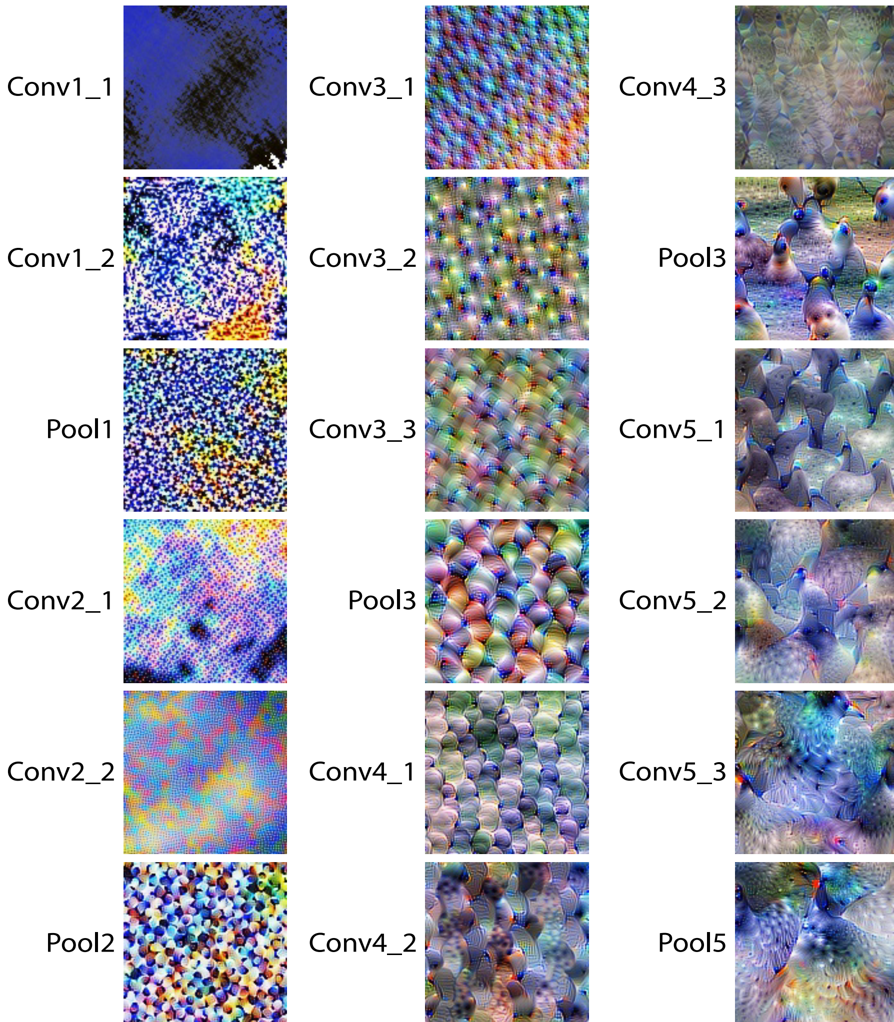


Figure 4.24: Example of visualizing for multiple filters combined. In this example the top 100 filters in each layer where used.

4.4.3 Applications and usefulness

Using Deep Taylor Decomposition as a guide for feature visualization is arguably slightly better than visualizing random filters. The networks we use are trained to recognize features for thousand different classes. By utilizing Deep Taylor Decomposition, we attempted to limit the number of features, and give a classification which is related to the results of feature visualization.

Visualizing single filters proved to be less useful than first anticipated. One can argue that features which exist in the original classification are visible in the resulting visualizations, but it is hard to find any conclusive evidence. Combining several relevant filters to create a visualization shows how a concept emerges throughout a CNN. The results are still hard to interpret, but they are to some degree relate to the image used to create the relevance scores. All in all, it seems like a good approach in order to be able to explain visualized features better, but there is still a lot of room for improvement.

Chapter 5

Conclusion

The goal of this thesis was to create a platform where a user could explore different convolutional neural networks by utilizing different visualization techniques for CNNs. We built a web-based platform to perform this task and presented results for each visualization technique we implemented. Each result was discussed individually, and we evaluated its usefulness in the context of achieving a better understanding of convolutional neural networks. In addition, we applied several visualization methods in order to evaluate the potential of combining visualization techniques.

Can we create a platform where users can apply different visualization techniques on networks with different architectures?
Creating a platform for visualizing convolutional networks was an achievable task. Generalizing for multiple networks, however, proved to be more complicated than first anticipated. Every visualization technique we implemented can be used independently of the network architecture, at least in theory. In practice, networks are assembled differently from architecture to architecture. An example of this is the Inception V1 network, which runs several convolutional operations in parallel. This feature made LRP more complicated and had to be solved by creating a unique module for this particular network. Another issue we encountered was the preprocessing needed for different network architectures. Different architectures may have different sizes for their input, and different preprocessing operations which have to be executed before the network can process an image. To sum up our experience, it is possible to visualize almost any given network architecture, but given the wide range of possible designs, it is a complicated task to create a general system which can visualize any network out-of-the-box.

Does utilizing visualization techniques yield any information which can be used to explain a neural networks behavior, and if so what does it tell us?
We limited ourselves to four methods of visualization; Activation Visualization,

Deep Taylor Decomposition, Feature Visualization and DeepDream. Each method tries to visualize the network in a unique way, but they all try to give a better understanding of artificial neural networks.

Visualizing activations in each layer shows which filters in a given layer have the highest activation values. It can be used to identify filters in a convolutional layer which react to certain features in an image. Another use-case for Activation Visualization is to identify "dead filters" which do not contribute to classifying images. One drawback of this method is how well it scales with deep neural networks. Deep neural networks often have hundreds of filters and output-classes which makes it hard to distinguish between a dead filter, and a filter only reacting to one particular feature.

Deep Taylor Decomposition was one of the more interesting visualization techniques, which had a lot of potential. This method excelled at highlighting features which were relevant for the generated classification. The heatmaps generated using this method can be used to identify which features a network uses to identify an object, and which parts of an object the network has learned to recognize. The method can also be used to reinforce claims to why an image received a wrong classification. Interpreting the highlighted features, and how they correspond to the generated classification is not always a trivial task, and is not guaranteed to be a useful tool, but in general it proved to be a good method for exploring CNNs.

Feature Visualization generated images that proved to be good indications of the underlying features within a trained CNN. By selecting a specific group of neurons for the visualization process, we were able to get a better understanding as to how information had been encoded into hidden layers during training. We were also able to see how different parts of the CNN interacted with each other to form new features. Applying various techniques, such as adding stochastic transformations and the introduction of alternate parameter-spaces showed a remarkable improvement in terms of comprehensibility. Compared to naive feature inversion with no image priors, the optimized images were much easier to recognize as actual objects and concepts.

DeepDream works similarly to feature visualization, but is run over an actual image. This made the algorithm enhance features already present in the picture. While there exist a few use-cases were this could come in handy to better understand a CNN, this method is probably better suited to be used in the field of computer-generated art.

Is there any additional information to be gained by combining different visualization techniques?
By combining Feature Visualization and Deep Taylor Decomposition, we attempted to create a bridge between the result of Feature Visualization and the different classes of a CNN. The idea was to make it easier to recognize features in the results of feature visualization by having a class which the features belong to. Using Deep Taylor Decomposition, we created a method for assigning a relevance score to each filter, depending on the classification generated by a CNN. The results were of

varying quality. The higher levels of the network seemed to correspond to random patterns, which were hard to link to the corresponding classification. In the lower layers, more abstract features appeared, which one could interpret as being related to the classification. By combining these visualization techniques we were able to create feature visualizations within a context which made them easier to interpret, but there still remains a lot of uncertainty and room for improvement.

We were able to create a platform which implemented several different visualization techniques into one system. By utilizing these visualization techniques, we explored two different convolutional neural networks and were able to better explain the correlation between an image and the classification generated by a CNN. We were also able to visualize the internal representation of features within a neural network, and to some extent improve the interpretability of the results.

5.1 Future work

Visualizing convolutional neural networks proved to be an interesting research area, with many possibilities. As we worked towards achieving our research goal we encountered many paths which seemed interesting to follow. In this section we list a few general areas which could potentially be interesting to research further.

5.1.1 Training convolutional neural networks

One of the most intriguing directions to follow would have been to train networks, and visualize them during and after training. The pretrained networks we worked with were able to distinguish between 1000 different classes. This meant that they had learned several different features for a wide range of distinct classes. By training our own networks, we could limit the number of different classes to a more manageable number, which could make it easier to interpret the results of feature visualization. In addition, we would have full control of all training parameters, which would include things such as learning rate and the dataset used to train the network. By controlling the dataset, we could train the same network on a range of different classes, and perform a comparison between networks trained on different datasets. Visualizing during training could be a possible method for monitoring the training process. It could also give insight into how representations within the network are formed.

5.1.2 Further improving the interpretability of feature visualization

While we have used a lot of different techniques in order to generate feature visualizations that were easy to interpret, there is still room for improvement in this

area. Including additional image priors to the optimization process could improve the interpretability even further. It should be noted that heuristics used to make results more natural-looking usually go against trying to optimize for activation values alone. The future work in this area should therefore also focus on keeping the essential aspects of the "raw" features present in the visualizations.

5.1.3 Smart selection of neuron-groups for feature visualization

As demonstrated throughout this thesis, visualizing individual filters inside a CNN will give us a better understanding of how the weights are able to encode complex information, and the relations between different elements within the hidden layers. Whenever a CNN performs a prediction, every single weight inside the network is playing a small role in the final output. There is reason to believe that groups of neurons, possibly spanning multiple layers, are able to represent more high-level concepts than the individual parts. Discovering smart ways to navigate the vast feature space of a deep convolutional network, selecting interesting groups of neurons to visualize could be an interesting area to explore further.

Bibliography

Alexander Mordvintsev, Christopher Olah and Mike Tyka (2015). *Inceptionism: Going Deeper into Neural Networks.*

Bach, Sebastian et al. (2015). "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation". In: *PloS one* 10.7, e0130140.

Castelvecchi, Davide (2016). "Can we open the black box of AI?" In: *Nature News* 538.7623, p. 20.

DeepDream - Wikipedia (n.d.). `https://en.wikipedia.org/wiki/DeepDream`. Accessed: 2018-05-28.

Deng, J. et al. (2009). "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09.*

Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285.*

Engilberge, M., E. Collins, and S. Süsstrunk (2017). "Color representation in deep neural networks". In: pp. 2786–2790.

Erhan, Dumitru et al. (2009). "Visualizing higher-layer features of a deep network". In: *University of Montreal* 1341.3, p. 1.

Gatys, Leon A, Alexander S Ecker, and Matthias Bethge (2015). "A neural algorithm of artistic style". In: *arXiv preprint arXiv:1508.06576.*

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep sparse rectifier neural networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323.

Goodfellow, Ian J, Jonathon Shlens, and Christian Szegedy (2014). "Explaining and harnessing adversarial examples". In: *arXiv preprint arXiv:1412.6572.*

Griffin, Gregory, Alex Holub, and Pietro Perona (2007). "Caltech-256 object category dataset". In:

Guo, Haitao, G. A. Sitton, and C. S. Burrus (1998). "The quick Fourier transform: an FFT based on symmetries". In: *IEEE Transactions on Signal Processing* 46.2, pp. 335–341.

Halkjær, Søren and Ole Winther (1997). "The effect of correlated input data on the dynamics of learning". In:

He, Kaiming et al. (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167*.

Khaligh-Razavi, Seyed-Mahdi and Nikolaus Kriegeskorte (2014). "Deep Supervised, but Not Unsupervised, Models May Explain IT Cortical Representation". In: *PLOS Computational Biology* 10, pp. 1–29. URL: https://doi.org/10.1371/journal.pcbi.1003915.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Krasner, Glenn E, Stephen T Pope, et al. (1988). "A description of the model-view-controller user interface paradigm in the smalltalk-80 system". In: *Journal of object oriented programming* 1.3, pp. 26–49.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., pp. 1097–1105.

Kuzovkin, Ilya et al. (2018). "Activations of Deep Convolutional Neural Network are Aligned with Gamma Band Activity of Human Visual Cortex". In: URL: https://www.biorxiv.org/content/early/2018/05/02/133694.

Lapuschkin, Sebastian et al. (2016). "The LRP toolbox for artificial neural networks". In: *The Journal of Machine Learning Research* 17.1, pp. 3938–3942.

Laurens van der Maaten and Geoffrey Hinton (2008). *Visualizing Data using t-SNE*.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Lin, Min, Qiang Chen, and Shuicheng Yan (2013). "Network In Network". In: *CoRR* abs/1312.4400. arXiv: 1312.4400. URL: http://arxiv.org/abs/1312.4400.

Mahendran, Aravindh and Andrea Vedaldi (2014). "Understanding Deep Image Representations by Inverting Them". In: *CoRR* abs/1412.0035. arXiv: 1412.0035. URL: http://arxiv.org/abs/1412.0035.

– (2015). "Visualizing Deep Convolutional Neural Networks Using Natural Pre-Images". In: *CoRR* abs/1512.02017. arXiv: 1512.02017. URL: http://arxiv.org/abs/1512.02017.

Martín Abadi et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Mohan, Rahul (2014). "Deep deconvolutional networks for scene parsing". In: *arXiv preprint arXiv:1411.4101*.

Montavon, Grégoire et al. (2017). "Explaining nonlinear classification decisions with deep taylor decomposition". In: *Pattern Recognition* 65, pp. 211–222.

Ng, Andrew Y (2004). "Feature selection, L 1 vs. L 2 regularization, and rotational invariance". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM, p. 78.

Nguyen, Anh Mai, Alexey Dosovitskiy, et al. (2016). "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks". In: *CoRR* abs/1605.09304. arXiv: 1605.09304. URL: http://arxiv.org/abs/1605.09304.

Nguyen, Anh Mai, Jason Yosinski, and Jeff Clune (2016). "Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks". In: *CoRR* abs/1602.03616. arXiv: 1602.03616. URL: http://arxiv.org/abs/1602.03616.

Noh, Hyeonwoo, Seunghoon Hong, and Bohyung Han (2015). "Learning deconvolution network for semantic segmentation". In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1520–1528.

Odena, Augustus, Vincent Dumoulin, and Chris Olah (2016). "Deconvolution and Checkerboard Artifacts". In: *Distill*. DOI: 10.23915/distill.00003. URL: http://distill.pub/2016/deconv-checkerboard.

Olah, Chris, Alexander Mordvintsev, and Ludwig Schubert (2017). "Feature Visualization". In: *Distill*. https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

Russakovsky, Olga et al. (2015). "Imagenet large scale visual recognition challenge". In: *International Journal of Computer Vision* 115.3, pp. 211–252.

Simonyan, Karen and Andrew Zisserman (2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556. arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

Steinkraus, Dave, I Buck, and PY Simard (2005). "Using GPUs for machine learning algorithms". In: *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*. IEEE, pp. 1115–1120.

Stutz, David (2014). "Understanding convolutional neural networks". In: *InSeminar Report, Fakultät für Mathematik, Informatik und Naturwissenschaften Lehr- und Forschungsgebiet Informatik VIII Computer Vision*.

Szegedy, Christian, Wei Liu, et al. (2015). "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Szegedy, Christian, Vincent Vanhoucke, et al. (2015). "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567. arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

Taylor, Luke and Geoff Nitschke (2017). "Improving Deep Learning using Generic Data Augmentation". In: *CoRR* abs/1708.06020. arXiv: 1708.06020. URL: http://arxiv.org/abs/1708.06020.

Wei, Donglai et al. (2015). "Understanding Intra-Class Knowledge Inside CNN". In: *CoRR* abs/1507.02379. arXiv: 1507.02379. URL: http://arxiv.org/abs/1507.02379.

Yosinski, Jason et al. (2015). "Understanding neural networks through deep visualization". In: *arXiv preprint arXiv:1506.06579*.

Yu, Wei et al. (2014). "Visualizing and comparing convolutional neural networks". In: *arXiv preprint arXiv:1412.6631*.

Zeiler, Matthew D and Rob Fergus (2014). "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer, pp. 818–833.

Zeiler, Matthew D, Graham W Taylor, and Rob Fergus (2011). “Adaptive deconvolutional networks for mid and high level feature learning”. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE, pp. 2018–2025.

Appendix A

Activation Visualization

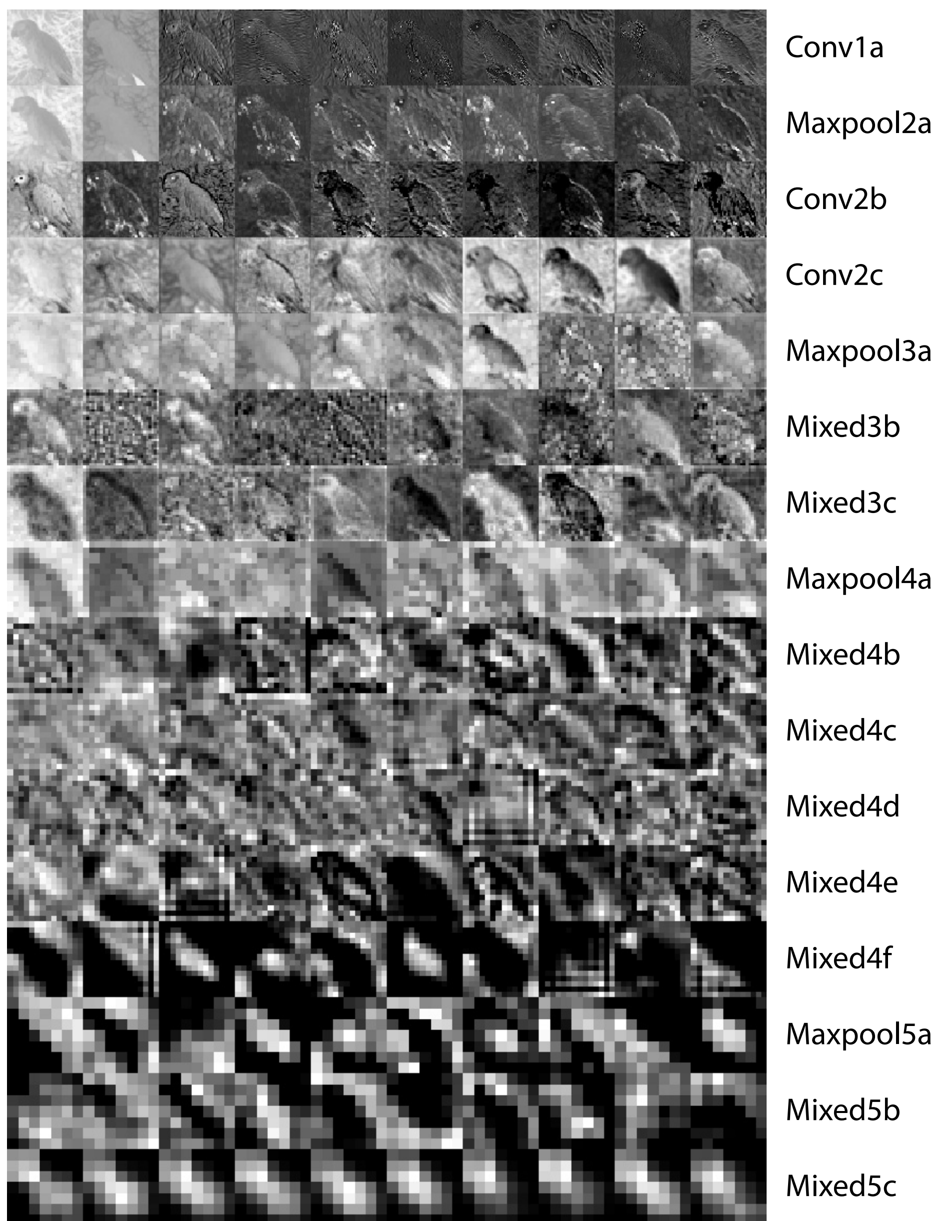


Figure A.1: Example showing the top 10 activations from all layers. The input contained an African Grey parrot and was classified correctly as such. The activations are taken from the Inception V1 network.

Appendix B

Visualization results

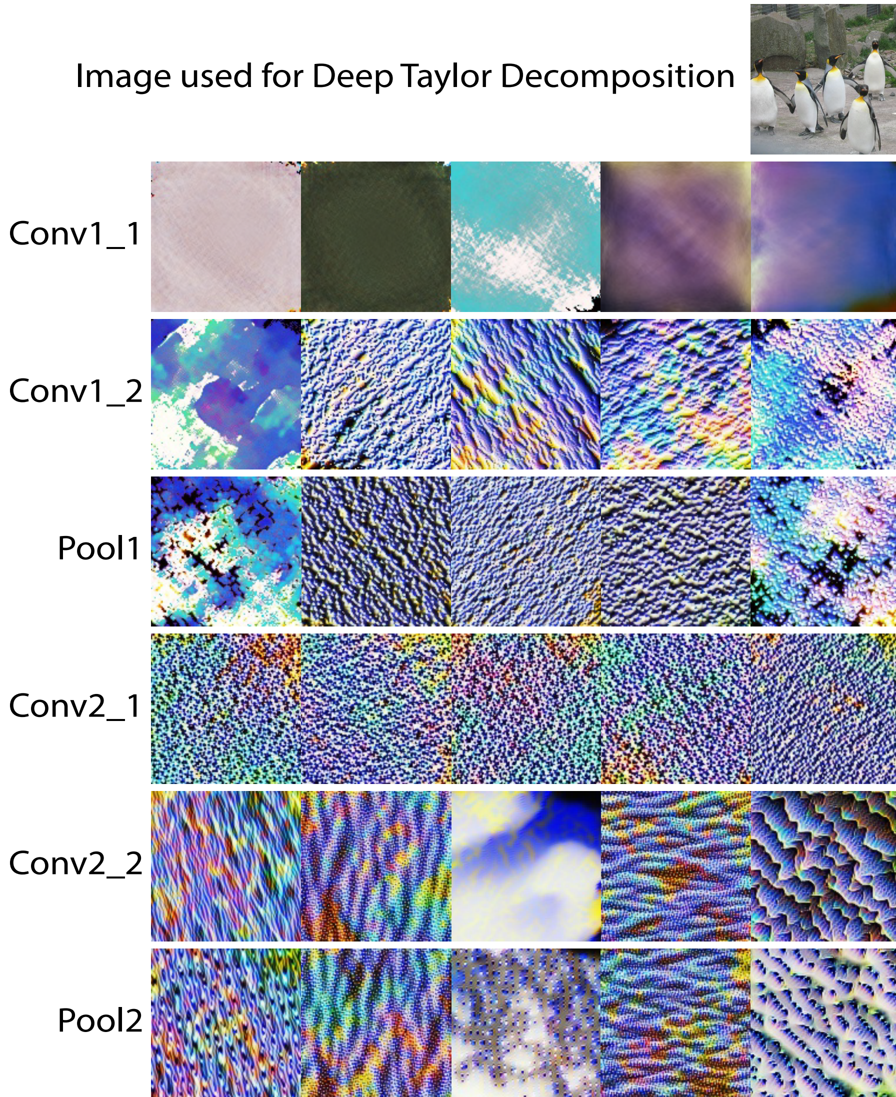


Figure B.1: Visualizing the top 5 filters with the highest relevance scores for layers 1-6 in the VGG-16 network.

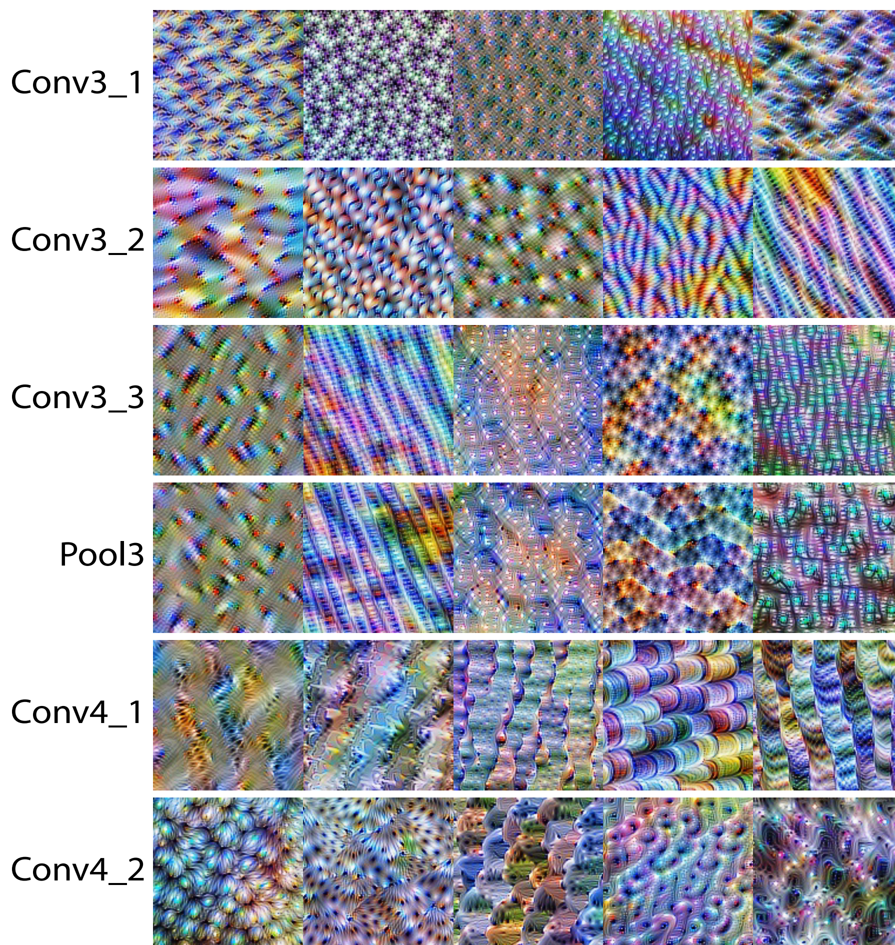


Figure B.2: Visualizing the top 5 filters with the highest relevance scores for layers 7-12 in the VGG-16 network.

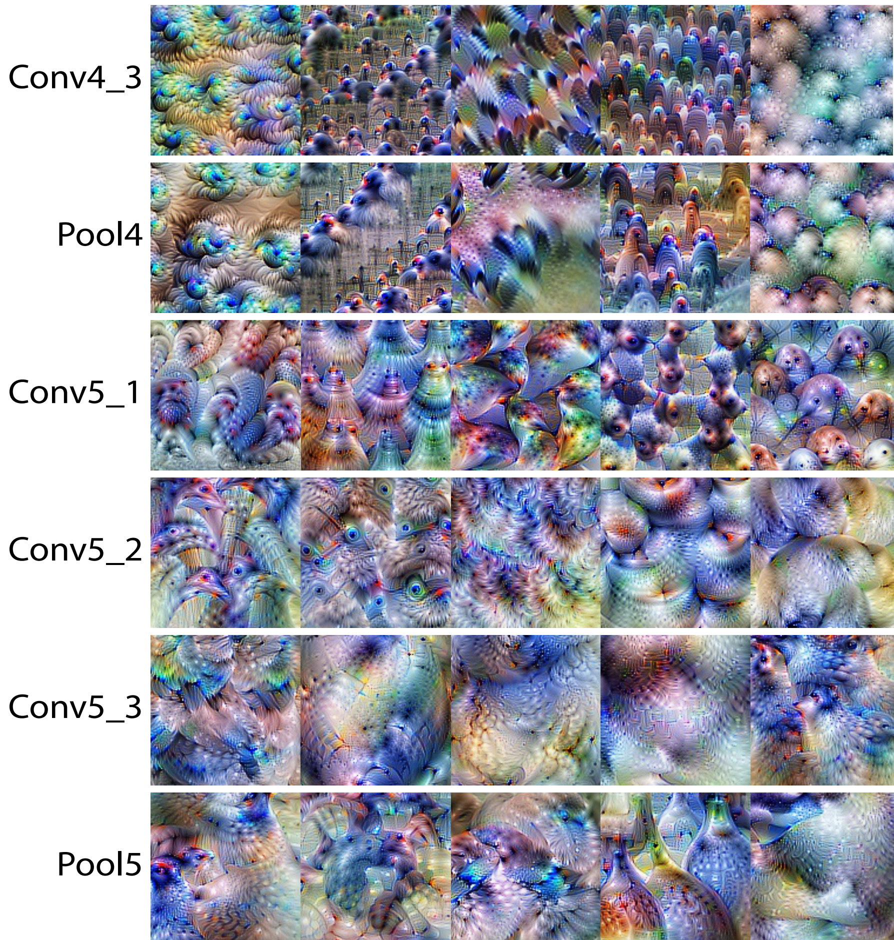


Figure B.3: Visualizing the top 5 filters with the highest relevance scores for layers 12-18 in the VGG-16 network.

Appendix C

Screenshots of user interface

This chapter contains a set of screenshots from the various pages within our platform, with the exception of the front page, which was displayed in Figure 3.8. In addition, a short caption is given, explaining what each screenshot contains, as well as the functionality of the selected page.

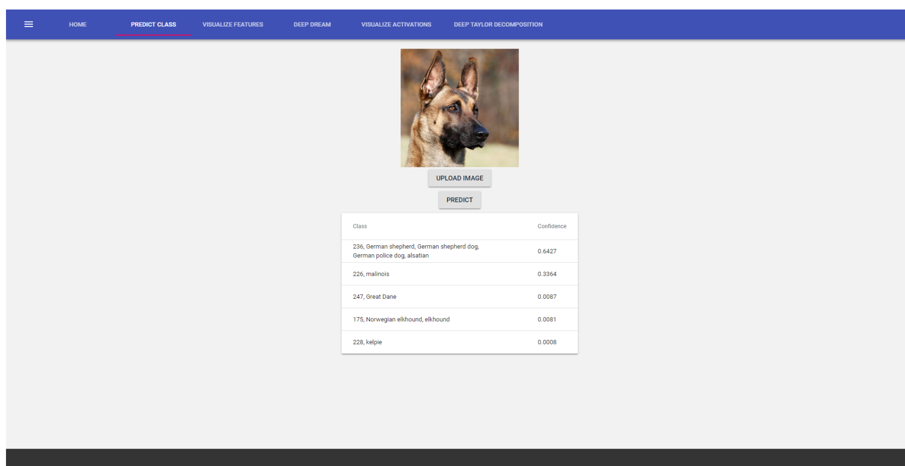


Figure C.1: Screenshot of the prediction page. Here a user can upload an image and make the network predict its classification. The top five results are displayed, along with the network's confidence for each class.

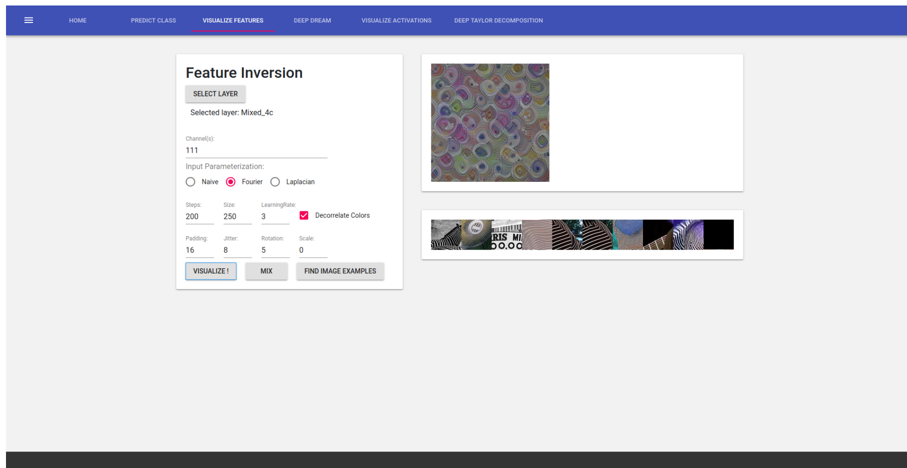


Figure C.2: Screenshot of the feature visualization page. The settings component is displayed on the left. It has three buttons at the bottom, for visualizing one or more features, visualizing a combination of features or fetching images resembling the feature. The results from the feature inversion appear in the top right, while the fetched images in the bottom right.

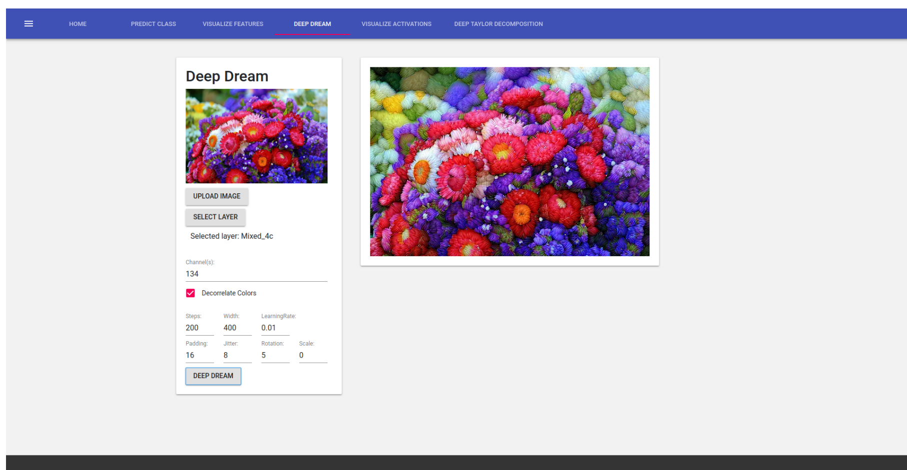


Figure C.3: Screenshot of the DeepDream page. A settings component for tuning parameters in the algorithm is displayed on the left, while the final result appears on the right side.

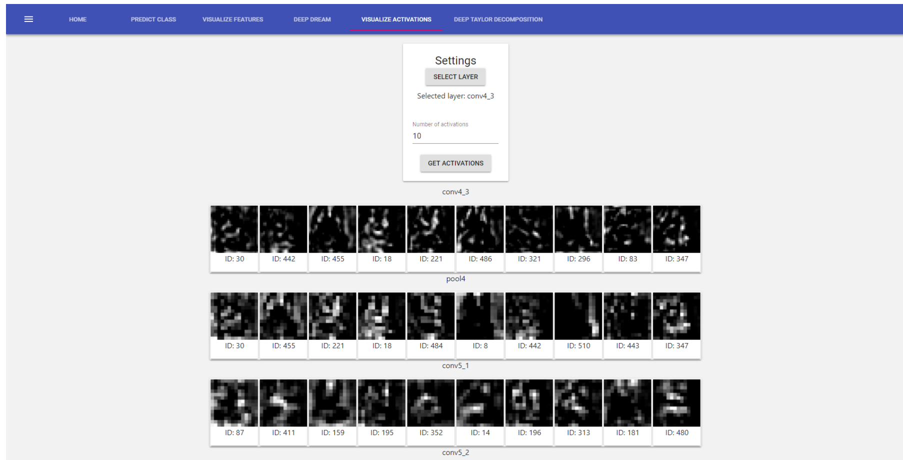


Figure C.4: Screenshot of the page for visualizing activations. In this example, the top ten activations for several layers are displayed, along with their ID, which translates to their index in the original matrix of output values.

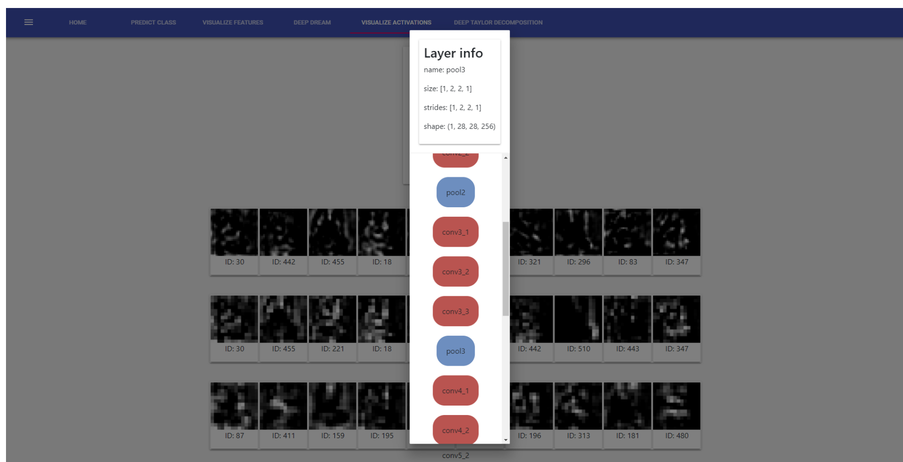


Figure C.5: Screenshot showing the module for selecting layers within the network. This module is used in Three pages; Activation Visualization, Feature Visualization and Deep-Dreaming. The user clicks a layer to select it. Afterwards, information about the selected layer, such as stride and output-dimensions are displayed at the top.

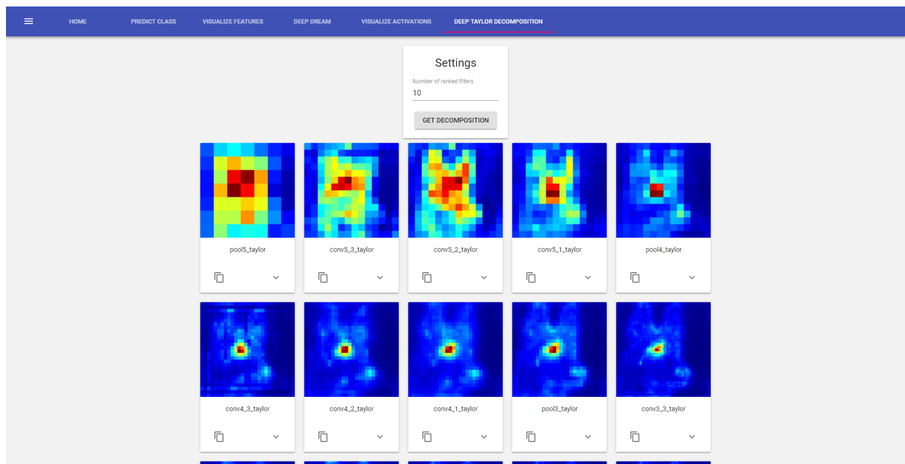


Figure C.6: Screenshot displaying the page for Deep Taylor Decomposition. The page displays one heatmap for each layer in the selected CNN. The original image can be seen and changed on the prediction page.