NTNU
Norwegian University of
Science and Technology

Web Interface for Deep Learning with Case Study in Facial Recognition

Annie Aasen
Mikael Bjerga

Master of Science in Computer Science

Submission date:	June 2017
Supervisor:	Theoharis Theoharis, IDI
Co-supervisor:	Aleksander Rognhaugen, IDI
	Igor Barros Barbosa, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

With the recent advancements in computational power, artificial neural networks are once again on the rise. However, the increase in depth and complexity further complicates researchers' ability to make sense of a network's behaviour. To that end, a number of visualization techniques have been developed, aiming to improve the understanding of artificial neural networks. Through visualization, researchers can gain insight into the inner workings of their networks. The images allow for easier identification of limitations and weaknesses, that can further lead to suggestions on improvements. Unfortunately, there exists a limited number of tools to facilitate the creation of such visualizations. This thesis presents a visualization tool that utilizes data produced by a network during training, to create visualizations using well-established visualization techniques. The main objective of the tool is to provide easy access to useful visualizations that can aid in understanding artificial neural networks, through a convenient API and web interface.

In this thesis, we have studied a number of visualization techniques to be implemented in our tool. We have also reviewed existing visualization tools for various machine learning libraries, to identify gaps in the offered options. To extend the options, we have implemented a visualization tool for Keras that combines the management of networks with advanced, real-time visualizations. To demonstrate the tool's capabilities, we include and interpret the visualizations created from two example networks.

In addition to the implemented visualization tool, the thesis also presents a case study in face recognition. The area of face recognition research continues to push boundaries, but still face difficulties with recognition in unconstrained environments. Obstacles like image quality, pose, illumination, and expressions present many challenges. While artificial neural networks have enjoyed much success in the field of face recognition, they still show room for improvement.

The thesis presents a novel method for improving face recognition systems using artificial neural networks, that utilizes information available in facial expressions instead of trying to overcome expressions through invariance. The case study examines three separate network architecture approaches: one with no utilization of expression information, one with an additional expression input, and one with an additional expression output. Examining their performance determined that the networks with an additional expression input performed better than their baseline counterparts. However, the networks with an additional expression output performed overall worse than the baseline networks. The findings indicate that the proposed extra input approach has further research appeal.

Sammendrag

Datamaskiner blir stadig kraftigere, og kunstige nevrale nettverk er igjen et populært forskningsfelt. Økningen i dypde og kompleksitet kompliserer imidlertid forskeres evne til å forstå hvordan et slikt nettverk oppfører seg. Som en følge av dette har en mengde visualiseringsteknikker blitt utviklet, med mål om å forbedre forståelsen av nevrale nettverk. Gjennom visualisering kan forskere få innsikt i hva som foregår inne i nettverkene sine. Bildene kan gjøre identifisering av svakheter lettere, som videre kan føre til forslag om forbedringer. Dessverre finnes det få verktøy for å forenkle bruken av slike visualiseringer. Denne oppgaven presenterer et verktøy som visualiserer treningsdataen som et nettverk produserer, ved hjelp av etablerte visualiseringsteknikker. Hovedformålet med verktøyet er å tilgjengeliggjøre nyttige visualiseringer som kan bidra til å forstå nevrale nettverk, gjennom et praktisk API og webgrensesnitt.

I denne oppgaven har vi studert en rekke visualiseringsteknikker for implementering i verktøyet. Vi har også gjennomgått eksisterende visualiseringsverktøy for ulike maskinlæringsbiblioteker, for å finne mangler i utvalget. For å utvide utvalget har vi implementert et visualiseringsverktøy for maskinlæringsbiblioteket Keras, som kombinerer administrasjon av nettverk med avanserte visualiseringer i sanntid. For å demonstrere verktøyets evner, har vi inkludert og tolket visualiseringer produsert ved hjelp av to eksempelnettverk.

I tillegg til det implementerte visualiseringsverktøyet presenterer denne oppgaven også en casestudie i ansiktsgjenkjenning. Teknologien for ansiktgjenkjenning utvikler seg stadig til det bedre, men støter fortsatt på problemer i møte med den virkelig verden. Variasjoner i bildekvalitet, positur, belysning og ansiktsuttrykk byr på store utfordringer. Selv om nevrale nettverk har hatt stor suksess innen ansiktsgjenkjenning, er det fortsatt rom for forbedring.

Oppgaven presenterer en ny metode for å forbedre systemer for ansiktgjenkjenning ved bruk av nevrale nettverk, som utnytter informasjonen i ansiktsuttrykk istedenfor å prøve å overkomme dem. Casestudiet undersøker tre separate tilnærminger for nettverksarkitektur: en uten bruk av ansiktsuttrykk, en med en ekstra inngangsnode for ansiktsuttrykk, og en med en ekstra utgangsnode for ansiktsuttrykk. Ved å sammenlikne resultatene, så vi at nettverkene med en ekstra inngangsnode gjorde det bedre enn tilsvarende standardnettverk, mens nettverkene med en ekstra utgangsnode alt i alt gjorde det dårligere enn tilsvarende standardnettverk. Dette indikerer at den foreslåtte tilnærmingen med en ekstra inngangsnode er interessant for videre forskning.

Project Description

This Master's thesis will focus on improving upon an interface for visualizing data produced by an artificial neural network during training. An initial version of this interface has been completed as part of the Specialization Project. The interface allows a user to upload and run a program that trains a network, and visualizes the data created by the program using various visualization techniques.

Also included in the thesis is a case study on the application of artificial neural networks in face recognition systems. Many networks struggle with overcoming the challenge of facial expressions, since the features of a face vary greatly with the expression it portrays. This thesis propose to exploit face expression data and explore how it can be utilized to improve a face recognition system.

Preface

This thesis was written over the course of the spring semester 2017 for the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

The work has been conducted with the guidance of several people. We would like to thank our main supervisor, Theoharis Theoharis, for taking the time away from his sabbatical to provide us with input on the project and report. Furthermore, we would like to thank his PhD-students Aleksander Rognhaugen and Igor Barros Barbosa for their co-supervision.

A special thank you to Nils Herdé for support and proofreading of the thesis.

Annie Aasen and Mikael Bjerga 19.06.2017

Contents

Abstract i

Sammendrag ii

Project Description iii

Preface v

Table of Contents x

List of Tables xi

List of Figures xiv

List of Code Snippets xv

Abbreviations xvii

1 Introduction 1

1.1 Background 1
1.2 Motivation 2
1.3 Contributions 3
1.4 Research Questions and Objectives 3
1.4.1 Visualization Tool 3
1.4.2 Case Study in Face Recognition 4
1.5 Research Methodology 4
1.5.1 Research Strategy 4
1.5.2 Data Generation Methods 5
1.5.3 Data Analysis 6
1.6 Thesis Structure 6

2 Background Theory **7**
2.1 Artificial Neural Networks . . . 7
2.1.1 Biological Background . . . 7
2.1.2 The Multi-Layer Perceptron . . . 8
2.1.3 Activation Functions . . . 8
2.1.4 Training a Network . . . 10
2.1.5 Gradient Descent and Backpropagation . . . 11
2.2 Convolutional Neural Networks . . . 12
2.2.1 Convolution Operation . . . 13
2.2.2 Convolution as Matrix Multiplication . . . 14
2.2.3 Convolutional Layer . . . 15
2.2.4 Hyperparameters in a Convolutional Layer . . . 16
2.2.5 Training a Convolutional Layer . . . 18
2.2.6 Transposed Convolutional Layer . . . 19
2.2.7 Max Pooling Layer . . . 19
2.3 Visualization . . . 20
2.3.1 Training Progress Measurements . . . 20
2.3.2 Layer Activations . . . 21
2.3.3 Saliency Maps . . . 21
2.3.4 Deconvolutional Network . . . 22
2.3.5 Deep Visualization . . . 24
2.4 Transfer Learning . . . 27
2.4.1 Fixed Feature Extractor . . . 28
2.4.2 Fine-Tuning . . . 28
2.5 Facial Expressions . . . 28
2.5.1 CNNs for Expression Recognition . . . 28
2.5.2 Overcoming Expressions in Face Recognition . . . 29

3 Related Work **31**
3.1 Visualization Tool . . . 31
3.1.1 Existing Visualization Tools . . . 31
3.1.2 Visualization Tools as Influential Factors . . . 33
3.2 Case Study in Face Recognition . . . 34
3.2.1 Improving Face Recognition . . . 34
3.2.2 Using ILSVRC Networks for Face Recognition . . . 35
3.2.3 Application to Case Study . . . 35

4 Implementation **37**
4.1 Visualization Tool . . . 37
4.1.1 Prototype . . . 37
4.1.2 System Development Methodology . . . 38
4.1.3 Focus Quality Attributes . . . 39
4.1.4 Technology Decisions . . . 40
4.1.5 Overview of the Architecture . . . 41
4.1.6 The Flask Application . . . 41
4.1.7 Database . . . 44

4.1.8 User Storage 44
4.1.9 Running a Python Script 45
4.1.10 Custom Keras Callbacks 46
4.1.11 The Bokeh Server 50
4.1.12 Bokeh Applications 51
4.2 Case Study in Face Recognition 53
4.2.1 Exploiting Expression Information 53
4.2.2 Experimental Network Idea 54
4.2.3 Method 54
4.2.4 Pretrained Model 55
4.2.5 Dataset and Preprocessing 56
4.2.6 Case Study Networks 58

5 Results **63**
5.1 Visualization Tool 63
5.1.1 Example Networks 63
5.1.2 MNIST Visualization Results 65
5.1.3 VGG Visualization Results 73
5.2 Case Study in Face Recognition 80
5.2.1 Network Performance 80
5.2.2 Training 80

6 Discussion **87**
6.1 Visualization Tool 87
6.1.1 Training Progress 87
6.1.2 Layer Activations 87
6.1.3 Saliency Maps 88
6.1.4 Deconvolutional Network 89
6.1.5 Deep Visualization 90
6.1.6 Usefulness of the Visualization Tool 92
6.2 Case Study in Face Recognition 92
6.2.1 Baseline Networks 93
6.2.2 Extra Input Networks 93
6.2.3 Extra Output Networks 93
6.2.4 Comparing Approaches 94
6.2.5 Experienced Limitations 95

7 Conclusion and Future Work **97**
7.1 Conclusion 97
7.2 Future Work 98
7.2.1 Visualization Tool 98
7.2.2 Case Study in Face Recognition 99

Bibliography **101**

A Installation & Setup **105**
A.1 Installation 105
A.2 Configuration and Setup 105
A.2.1 Linux and MacOS 106
A.2.2 Windows 106
A.3 Starting the Visualization Tool 106
A.3.1 Linux and MacOS 106
A.3.2 Windows 106

B User Manual **107**

C Callbacks API **117**
C.1 Usage of Callbacks 117
C.1.1 CustomCallbacks 118
C.2 Available Callbacks 119
C.2.1 NetworkSaver 119
C.2.2 TrainingProgress 119
C.2.3 LayerActivations 119
C.2.4 SaliencyMaps 120
C.2.5 DeconvolutionalNetwork 120
C.2.6 DeepVisualization 121

D Visualization Files **125**
D.1 Training Progress 125
D.2 Layer Activations 126
D.3 Saliency Maps 126
D.4 Deconvolution Network 126
D.5 Deep Visualization 127

List of Tables

1.1 Overview of the thesis structure 6

2.1 Hyperparameters found in random search. 27

4.1 Pretrained network architecture 57
4.2 Baseline architecture configurations 59
4.3 Extra input architecture configurations 61
4.4 Extra output architecture configurations 62

5.1 Performance of case study networks with a single output 85
5.2 Identification performance of extra output case study networks 85
5.3 Expression performance of extra output case study networks 85
5.4 Configuration training times 86

List of Figures

1.1 Research Process Model 5

2.1 Biological neuron [1] 8
2.2 Mathematical model of a neuron [1] 8
2.3 The most commonly used activation functions 9
2.4 An example of 2D cross-correlation. 14

4.1 Overall Architecture of the System 42
4.2 Package Diagram 42
4.3 Flask project file structure 43
4.4 ER-diagram of the database 44
4.5 User storage structure 45
4.6 Sequence diagram of getting a visualization from Bokeh Server 52
4.7 Images from IMFDB 58

5.1 Visualization input images for MNIST 65
5.2 Visualization input images for VGG 66
5.3 Architecture of the MNIST network with layer names and types. Note that the layer type refers to the Keras 2 API. 67
5.4 Architecture of the VGG network with layer names and types. Note that the layers refer to the Keras 2 API. 67
5.5 Batch accuracy and loss for MNIST network 68
5.6 Layer activations for all MNIST layers 70
5.7 Change in activations in `conv2` of MNIST 71
5.8 Saliency maps for MNIST 72
5.9 Deconvolutional results for MNIST `pool` layer 73
5.10 Deep visualization from MNIST output units 74
5.11 Deep visualization of MNIST layer `fc2` 75
5.12 Deep visualization of MNIST layer`fc1` 75
5.13 Layer activations for VGG `block1_conv2` layer 75
5.14 Saliency maps for VGG 76

5.15 Deconvolution for VGG `block5_pool` layer 77
5.16 Comparison of deconvolution for VGG `block5_pool` layer 78
5.17 Comparison of deconvolutional visualizations for VGG `block3_pool` layer 79
5.18 Deconvolution for VGG `block1_pool` layer 79
5.19 Deep visualization of VGG `output` layer 81
5.20 Variations in deep visualization of VGG `output` layer 82
5.21 Hyperparameter variations in deep visualization of VGG `output` layer . 82
5.22 Deep visualization of VGG `fc2` layer 83
5.23 Deep visualization of VGG `fc1` layer 83
5.24 Deep visualization of VGG pooling layers in block 2-5 84
5.25 Learning rate variations in deep visualization of VGG `block5_pool` layer 84
5.26 Deep visualization of VGG `block1_pool` layer 84

B.1 Login 107
B.2 Creating a new user 108
B.3 Uploaded files 108
B.4 Uploading a file 109
B.5 File overview 110
B.6 Running a file 110
B.7 File overview when file is running 111
B.8 Output of a running file 111
B.9 Training progress 112
B.10 Layer activations part 1 113
B.11 Layer activations part 2 113
B.12 Saliency map 114
B.13 Deconvolutional network 114
B.14 Deep visualization 115

List of Code

2.1 Vanilla gradient descent 11
4.2 View for creating a user 44
4.3 Running a Python script using subprocess 46
4.4 Wrapper class for custom callbacks 47
4.5 Example implementation of a callback 49
4.6 Embedding the Bokeh server in Flask 51
4.7 Getting the data from the visualization files 53

Abbreviations

ANN artificial neural network. 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 15, 20, 21, 22, 25, 29, 31, 32, 34, 35, 38, 39, 40, 46, 53, 55, 63, 64, 92, 97, 98

API application programming interface. xiii, 2, 6, 33, 39, 40, 50, 53, 56, 59, 60, 62, 65, 97

CNN convolutional neural network. viii, 7, 12, 13, 16, 19, 20, 27, 28, 29, 34, 55, 63, 64, 87

DIGITS Deep Learning GPU Training System. 31, 32, 33

FACS Facial Action Coding System. 28

ILSVRC ImageNet Large Scale Visual Recognition Challenge. viii, 34, 35, 50, 55, 63, 64

IMFDB Indian Movie Face Database. xiii, 56, 57, 92

LFW Labeled Faces in the Wild. 34, 35

MLP multi-layer perceptron. 8

MNIST Modified National Institute of Standards and Technology. ix, xiii, 63, 64, 65, 69, 87, 88, 89, 90

PReLU Parametric Rectified Linear Unit. 9

ReLU Rectified Linear Unit. 8, 9, 16, 21, 22, 23, 56, 59, 60, 62

SGD Stochastic Gradient Descent. 11, 12

VGG Visual Geometry Group. ix, xiii, xiv, 35, 55, 56, 63, 64, 65, 69, 73, 80, 87, 88, 89, 90

Chapter 1

Introduction

This chapter will introduce the thesis by providing a background for the problem and research area. It will also explain the motivation behind the thesis, and what we aspire to contribute to the field, before composing the problem into a set of research questions and objectives. The research methodology will be explained in terms of the research strategy, data generation methods and data analysis. Finally, the chapter will describe the structure of the thesis.

1.1 Background

The thesis is concerned with artificial neural networks (ANNs) and the use of visualization techniques to gain an understanding of their behaviour. Furthermore, the thesis explores a novel approach to improve the accuracy of facial recognition networks. The work presented in the thesis is a continuation from the results of the Specialization Project.

The expected outcome of the thesis can be divided into two parts:

1. A visualization tool for deep learning
2. A case study in facial recognition

The first part proposes a tool for visualizing data produced during the training of an ANN. This visualization tool should be implemented in the form of a web interface where users can upload and run their Python scripts for creating and training such networks. The tool should use established visualization techniques to generate visualizations for the user, in order to help them gain insight into their networks and why their networks behave as they do. A prototype of this interface has been implemented as part of the Specialization Project and will serve as a foundation for the continuation of the development. A description of the prototype, as well as an overview of the implementation details for the thesis, will be presented in Chapter 4.

The case study in the second part explores whether it is possible to exploit facial expression data in order to improve an ANN for face recognition. The idea is that a network with access to facial expression data could learn the alterations of features associated with various expressions, and apply this to recognize faces which differ in the corresponding features. To examine the idea, a well known ANN architecture should be adapted to the face recognition domain to create three different networks: one conventional to act as a basis to be evaluated against and two experimental that exploit facial expression data. In the experimental networks, one will use the expression data as an additional input, while the other will have an extra expression output.

1.2 Motivation

In the modern computer science landscape, ANNs are on the rise. They have enjoyed considerable success in a plethora of fields and there exist a multitude of frameworks to help create and train such networks. Among the frameworks are TensorFlow[1] and Caffe2[2], which are open-source projects backed by tech giants Google and Facebook, respectively. Other popular frameworks are Theano[3], MXNet[4], Torch[5], and PyTorch[6]. However, an issue with ANNs is that they have been notoriously difficult to understand. Being dissatisfied with this knowledge vacuum, researchers have developed a number of visualization techniques to help gain insight into the inner workings of a network. Unfortunately, unlike the building of networks, there does not exist an extensive amount of options when it comes to facilitating the creation of visualizations. To combat this shortage, we aim to design and develop a tool that provides researchers with easy access to visualization techniques through a convenient user interface. More specifically, we plan to fill the void of an advanced visualization tool for the machine learning frameworks Theano[7] and TensorFlow through the use of Keras[8], a high-level application programming interface (API) supporting both frameworks. The tool would aid researchers using these popular frameworks to gain a better understanding of their networks and thus enabling further research.

One of the fields where ANNs have been successfully applied is the area of face recognition. While notable performance has been achieved, research in the area of face recognition is not complete. Achieving robust facial recognition could provide an effective, non-intrusive biometric identification scheme that could be used for both information and physical security. Such identification systems would allow users to identify themselves with ease, without the use of passwords, PINs or ID cards, and gives organizations a hard-to-forge, human-independent security measure. Examples include computer logins, ATM access, airport security, and building access [2, 3, 4]. We will therefore carry out a case

[1]https://www.tensorflow.org/
[2]https://caffe2.ai/
[3]http://deeplearning.net/software/theano/
[4]http://mxnet.io/
[5]http://torch.ch/
[6]http://pytorch.org/
[7]http://deeplearning.net/software/theano/
[8]https://keras.io/

study in the face recognition field, in addition to developing the visualization tool. We postulate that an approach that utilizes the information available in facial expressions may experience greater accuracy than one that tries to overcome expressions through invariance.

1.3 Contributions

The thesis introduces a visualization tool for Keras networks, implemented as a web interface, that can help researchers gain a better understanding of the behaviour of their ANNs. The effort will be concentrated on generating visualizations for networks whose input data reside in the image domain. The tool should be easily extensible, in regard to adding new visualization techniques and customizing the tool to a different machine learning framework. At the end of development, the visualization tool will be released as an open-source project[9].

The thesis also explores the face recognition field by investigating if the exploitation of facial expression data in face recognition systems could be beneficial. The results of the case study will indicate whether the approach has initial merit and should be researched further.

1.4 Research Questions and Objectives

This section condenses the problem stated into two research questions that will guide the problem solving process. We have also defined a set of research objectives that identifies the goals associated with each of the research questions. These goals must be achieved in order to be able to answer the questions.

1.4.1 Visualization Tool

We intend to aid researchers in the process of creating and training their ANNs. The research question is formulated as follows:

RQ 1: How can we develop a visualization tool to improve the understanding of the behaviour of an ANN?

To answer this question, we will need to complete the following research objectives:

RO 1.1: Examine existing visualization tools in order to uncover shortcomings.

RO 1.2: Identify visualization techniques that can be used to improve the understanding of an ANN.

RO 1.3: Develop a tool that incorporates the techniques found in 1.2 and addresses the shortcomings identified in 1.1.

[9]Found at https://github.com/mikaelbj/training-visualizer

1.4.2 Case Study in Face Recognition

We aim to explore whether it is possible to exploit facial expression data in order to improve an ANN for face recognition. The research question is formulated as follows:

RQ 2: How can facial expression data be utilized to improve a face recognition ANN?

To answer this question, we will need to complete the following research objective:

RO 2.1: Investigate how performance is affected when incorporating expression data in a face recognition system.

1.5 Research Methodology

This section will explain the research method used in our thesis. Note that this is not to be confused with the more specific system development method, which is the explanation and documentation of how the suggested systems are implemented. The system development method will be addressed in Chapter 4. The research methodology is the combination of research strategies and data generation methods to be used in the thesis. Even though our research consists of two separate problems, we plan to address them simultaneously, and will employ similar methods for both. Thus, parts of this section discuss the research in general instead of separately for each research question.

1.5.1 Research Strategy

Oates [5] proposed six different strategies for answering research questions. Since both of our questions are concerned with the implementation of some kind of computer system, it is natural to employ the design and creation strategy, which focuses on developing new IT products, or so-called artefacts. March & Smith [6] defines the final type of IT artefacts as instantiations. Instantiations are working systems demonstrating that constructs, models, methods, ideas or theories can be implemented in a computer-based system. The research output in our case will be two instantiations: the implemented visualization tool, and an implementation of a face recognition system incorporating a novel idea.

Vaishnavi & Kuechler [7] present an iterative process typical for design and creation, illustrated in Figure 1.1. The process suggests a "learning through making" approach and involves five steps, followed in an iterative cycle: awareness of problem, suggestion, development, evaluation and conclusion. The first step is to study existing literature and research to uncover the need for something. The second step provides a creative leap from the need to a simple idea of a solution. Then, the idea is implemented in the third step, followed by an evaluation of this implementation and an explanation of possible deviations from the expectation in the fourth step. Finally, in the fifth step, the result is documented, and the knowledge gained, as well as any possible future work, is identified.

For the visualization tool, the first couple of iterations of this cycle have already been conducted in the Specialization Project. The result of this project included many pointers to

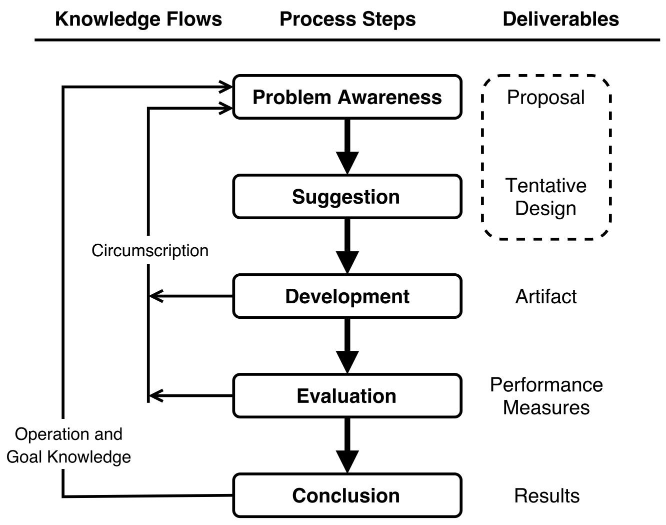


Figure 1.1: Research Process Model

future work, which will be addressed in this thesis. We will continue to repeat the process until we can conclude with an implementation that is satisfactory according to our research question and objectives. For the case study, however, no development was conducted in the Specialization Project. The focus was rather on the first two steps, namely to define the problem and suggest a solution. In this thesis, we will continue the process by implementing a suggested solution, and continue to evaluate and improve it until we reach a conclusion.

1.5.2 Data Generation Methods

The data that will be used in the thesis will mainly be qualitative data in the form of documents. For the implementation of the visualization tool, articles on various visualization techniques that can aid in understanding ANNs will be essential. We will also make use of documentation of the frameworks and technologies used in implementation. To document the implemented tool and capture our research strategy, we will generate our own documents using architectural diagrams and user manuals.

For the case study, we will need a large amount of images. Since our case study is concerned with the field of face recognition with expressions, we require images of people with different facial expressions, photographed under various conditions, that are labeled with identification and expression. There exist several datasets created solely for the purpose of face recognition, but the challenge will be to find one of appropriate size, containing the required labels. Unfortunately, the larger datasets examined have a tendency to be expensive or lack availability. The chosen dataset will be used to both train the networks and validate their performance afterwards.

1.5.3 Data Analysis

To present the results of the visualization tool, we will make use of well known ANN architectures, and show the visualizations produced at various stages of training. As the results are images, we will need to perform quantitative data analysis on these images. The goal will be to identify connections between the visualizations and the behaviour of the networks, both in terms of why they may perform well and why they may struggle.

Qualitative data analysis will be performed on the results of the case study. The prediction accuracy of an ANN is defined as the percent of correctly classified inputs, in our case images. The accuracy of our implemented network incorporating the proposed modification will be measured against the accuracy of the baseline network. By comparing these, we should be able to determine whether exploiting expressions in such a way could be beneficial.

1.6 Thesis Structure

Table 1.1 presents an overview of the structure of the thesis.

Chapter/Appendix	**Description**
1. Introduction	An overview of the research to be done in the thesis.
2. Background Theory	The theory relevant for the thesis.
3. Related Work	An overview of related work.
4. Implementation	The implementation of the visualization tool and case study in terms of methods and architecture.
5. Results	The results from both the visualization tool and the case study.
6. Discussion	Discussion of the results presented in the results chapter.
7. Conclusion and Future Work	A short conclusion and some pointers to future work.
A. Installation & Setup	A step by step guide on how to install and set up the visualization tool.
B. User Manual	How to use the visualization tool, including screenshots of the web interface.
C. Callbacks API	An API for the callbacks to be used with the visualization tool.
D. Visualization Files	A note on the format of the content of the visualization files.

Table 1.1: Overview of the thesis structure

Chapter 2

Background Theory

This chapter provides the background theory necessary for understanding the foundation of the work to be conducted in the thesis. Note that the topics here are not exhaustively explained. Only the parts relevant to the thesis is covered. We start with an introduction to ANNs and continue with a deeper look at convolutional neural networks (CNNs). Then, we describe several visualization techniques for visualizing such networks, followed by a brief overview of transfer learning, before ending the chapter with an examination of facial expressions and their relation to CNNs.

2.1 Artificial Neural Networks

In this section, the basic concepts of ANNs will be explained. First, some information on their biological background is presented, before describing the structure of a simple ANN. Then, common activation functions are explained, followed by a section on the training process of an ANN. Finally, the gradient descent and backpropagation processes involved in the training will be covered in more detail.

2.1.1 Biological Background

The idea of an ANN is that it can approximate any continuous function, and that it has the possibility of learning. ANNs are inspired by the structure and behavior of a biological brain and its ability to learn, but they are generally not intended to be realistic models. A neural network consists of interconnected nodes referred to as units. These correspond to a biological neuron, the basic computational unit of the brain, structured as illustrated in Figure 2.1. The connections between the units corresponds to a biological neuron's dendrites, which provide input signals, and its single axon, which produces output signals and is connected to other neurons.

To simulate these signals, an ANN uses a mathematical model as illustrated in Figure 2.2, where the signal is multiplied by the weight of a connection. The weight of a specific

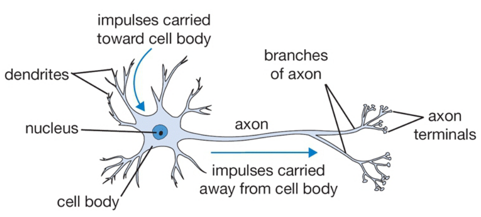

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

Figure 2.1: Biological neuron [1]

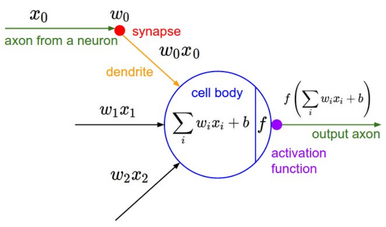


Figure 2.2: Mathematical model of a neuron [1]

connection controls how much the unit on one end influences the unit on the other end. The sum of all input signals are computed at the cell body of the unit, and a constant, b, is added from a bias unit. A bias unit is another unit that influences the output without interacting with the actual data. The unit fires an output signal determined by its activation function by performing a specific mathematical operation on the result.

2.1.2 The Multi-Layer Perceptron

The simplest form of an ANN is the single layer perceptron, which consists of only one layer of output units. This network can only learn linearly separable patterns. To allow the learning of complex patterns, more layers need to be added, resulting in the multi-layer perceptron (MLP). As layers are added, the depth of the network increases. This is the reason why the use of MLPs is referred to as ”deep learning”.

There are several different classes and types of ANNs. We will focus on the standard feed-forward neural network, and thus we will use the term ANN when referring to such a network. In a feed-forward network, all connections are directed from one layer to the next layer, and there are no cycles or connections between units in the same layer.

The first layer of an ANN is called the input layer. Similarly, the last layer is called the output layer. All layers in between are referred to as hidden layers, and are usually fully connected layers. This means that each layer node is connected to every node in the preceding layer. A layer can also be a convolutional layer or a pooling layer, but these cases will be covered later, in Section 2.2.

2.1.3 Activation Functions

The common behaviour of all activation functions is that they define the output of a unit given a set of inputs. There are a great variety of activation functions in use, but we will only describe those most commonly used and thus considered in our implementation. The mathematical function of three of these are plotted in Figure 2.3.

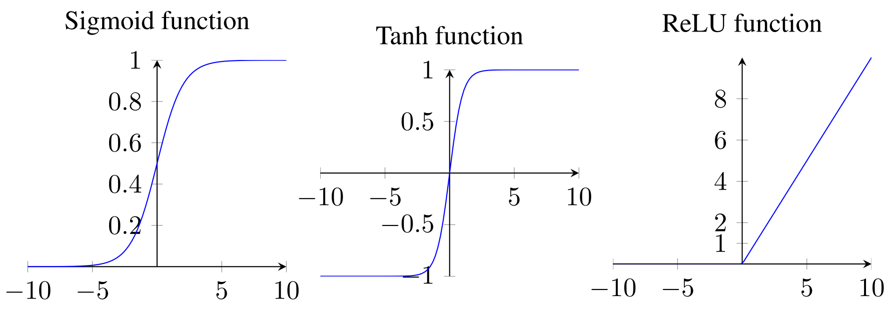


Figure 2.3: The most commonly used activation functions

Sigmoid

The sigmoid activation function is mathematically defined as $\sigma(x) = 1/(1 + e^{-x})$. The function takes a number as input, and outputs a number within a continuous range from 0 to 1. It is commonly used as the activation function for the output node of a binary classification problem. Apart from this, the other activations functions tend to be preferred to the sigmoid function because of its drawbacks [8]. The first drawback is that the outputs are not centered around zero, which can cause undesirable behaviour during learning. A bigger drawback is that the activation saturates, meaning that the units output mostly 0 and 1 instead of anything in between, and thus the gradient at these regions is very low. If the gradient falls to zero, the network will stop learning.

Tanh

The tanh activation function is a scaled version of the sigmoid function. Its mathematical function is $tanh(x) = 2\sigma(2x) - 1$. The scaling causes the tanh function to output numbers within the range of -1 and 1. This makes the output zero-centered, thus avoiding the first drawback of the sigmoid function. Even though the tanh function still suffers from saturation, it is preferred to the sigmoid function [8].

ReLU

The Rectified Linear Unit (ReLU) [9] activation has the mathematical function $f(x) = max(0, x)$. This means that it is close to linear; thresholding the input at zero, but keeping any value above. It has become highly popular, and is in general the activation function of choice [8]. It is non-saturating, and has been found to speed up the convergence of the learning. The computation of the function is also inexpensive compared to the sigmoid and tanh functions. The sole problem of the ReLU function is the possibility of ”dead” units, i.e. units that will never activate. Generally, adjusting the learning parameters can prevent this, but there have also been several attempts to fix this issue, for instance using so-called Parametric Rectified Linear Unit [10] or Maxout [11] units.

Softmax

The softmax activation function is a variant of the sigmoid function [12]. Instead of binary classification, it is used for the output units of a multiclass classification problem, where the goal is to classify instances into one of more than two classes. The mathematical equation for the softmax function is $f(x) = e^x / \sum_{j=0}^{k} e^{x_j}$. It calculates the probability distribution of each target class over all possible target classes. A convenient result of this is that the sum of all probabilities will be equal to one. Note that the softmax function is not used in hidden units, but only as the activation function of output units.

2.1.4 Training a Network

The weights of an ANN, often referred to as the parameters, can be learned by training the network on a set of training data in a procedure called supervised learning. Typically, the weights are initialized to small, randomly generated numbers. The training data consists of training pairs of input and output data, with the output being the desired output of the ANN for the given input. In classification, for instance, the input could be an image, and the output could denote what the image represents, e.g. a cat. In training, the difference between the desired output and the network's actual output is used to update its weights. This process is elaborated upon in Section 2.1.5. Often, it is useful to include a validation set and a test set in addition to the training set. Both of them should be representative of the training set. The validation set can be used during training to measure performance on data the network has not been explicitly trained on. The test set is used on fully trained networks to compare their performance. It is important that the different data sets consist of a substantial amount of diverse training pairs.

Data Preprocessing

A data set could be significantly improved by employing preprocessing methods. A common form of data preprocessing that we will employ is mean subtraction. It involves zero-centering the data by subtracting the mean across each individual feature, to ensure that the features are in a similar range. For images, this would be the values of the R, G and B channels. Note that the mean should only be computed from the training data, but it still has to be applied to the validation and test data.

Overfitting

The training data of an ANN can often be noisy and incomplete. Overfitting is when the ANN learns the details and noise of the data too well, making the network more specialized and negatively impacting its general performance. The phenomenon could be detected by examining the validation set during training. If the loss of the training set continues to decrease, but the loss of the validation set rises, this could suggest that the network is experiencing overfitting. An effective and simple regularization technique to prevent overfitting is dropout, where randomly selected units are ignored during training [13]. A unit is ”dropped out” with a given probability each weight update cycle. The technique makes the network less sensitive to specific weights, and results in a more generalized network that is less likely to overfit to the training data.

Loss Function

The goal of an ANN is to find a function that solves a certain problem, or more specifically, finds an optimal solution to the problem. The definition of this optimal solution is that given a loss function, there are no other solution with a lower loss value. The loss function is in that sense a measure of how far away a solution is from an optimal solution. The goal of the learning process is thus to find a function with the smallest possible loss. There are several loss functions that can be used in ANNs, each suited for a different purpose. For classification tasks where you want to minimize the number of misclassified training pairs, cross-entropy is the natural choice [14].

2.1.5 Gradient Descent and Backpropagation

To optimize the loss, we can use an algorithm called gradient descent, which minimizes any function iteratively. In short, the algorithm makes the learning process a repeated loop of evaluating the gradient, and then updating the weights based on this gradient. The simplest form of this algorithm, also called the vanilla version, is shown in Code 2.1.

```python
for i in range(epochs):
  weights_grad = evaluate_gradient(loss_function, data, weights)
  weights += -learning_rate * weights_grad
```

Code 2.1: Vanilla gradient descent

First, the algorithm evaluates the gradient of the loss function for the whole training data set with respect to the current weights. This is done using an algorithm called backpropagation. Often, the term is used to refer to the complete process of repeatedly computing the gradient and updating the weights. Each iteration of computing the gradient involves two steps: forward propagation and backward propagation, also known as forward pass and backward pass, respectively. The forward propagation is simply the propagation of an input through the network in order to generate an output. In the backward propagation step, the loss between the generated output and the desired output is computed, and the error is propagated backwards to the preceding layer. This layer then similarly computes its loss and error and propagates it backwards. The procedure is repeated in every layer to obtain an error value for each. The error values are then used to calculate the weight gradients. The next step of gradient descent is to adjust the weights, based on the evaluated gradient and a chosen learning rate. The learning rate controls the magnitude of the update, and is a crucial network hyperparameter. This process is repeated for a predefined number of epochs.

Stochastic Gradient Descent

In the vanilla version of gradient descent, the gradient is computed over the entire training data set, which is slow and expensive in terms of memory. There exist a number of different ways to optimize gradient descent by extending the vanilla version. Stochastic Gradient Descent (SGD) represents one of these optimizations. In this case, the gradient is computed over batches of the training data instead of over the whole set. Even though SGD is much more efficient than the vanilla version, it also has its drawbacks and therefore a number of improved algorithms have been developed.

Adding Momentum

One way to improve SGD is to add momentum. SGD can get stuck in local optima, and adding momentum can help it accelerate in the right direction. A further improvement is the Nesterov momentum, which adds some notion of where SGD is heading by approximating the next position [15].

Adaptive Learning Rate Methods

Adaptive learning rate methods are algorithms that automatically adapts the learning rate to the parameters. They are all quite similar, and perform thereafter. Ruder [15] presents the various methods, and we will briefly review his findings. In general, SGD often finds a minimum, but these methods can help to achieve convergence faster, and requires no tuning of the learning rate. The algorithms are usually implemented with a decent default starting value. Adagrad was the first adaptive learning rate method proposed, and the later additions are extensions of it. Its weakness is that the learning rate decreases to eventually become too small. AdaDelta and RMSprop are both methods that overcome this flaw, and they only differ slightly in implementation. Another option, Adam, adds momentum to the optimization as well.

2.2 Convolutional Neural Networks

The theory presented in this section is gathered from Goodfellow et al. [16], if not explicitly stated otherwise.

A variation of the traditional ANN, CNNs are especially well suited for handling input data that can be arranged in a grid-like topology. These types of networks can be applied to input of any dimension, but are commonly used in situations where the processing of images is required. We will concentrate on the convolution of images, as it is of most relevance to the thesis.

A typical CNN architecture involves multiple convolutional layers as the main body of the network, interspersed with pooling layers, before ending with one or more fully connected layers. A convolutional layer employs a mathematical operation called convolution, from which the name of both the layer and the network type is derived. Before explaining the convolutional layer and the pooling layer, as well as a convolutional layer variant called the transposed convolutional layer, we review the convolution operation and its matrix multiplication counterpart.

2.2.1 Convolution Operation

Generally, convolution can be defined as an operation on two functions, x and w, of a real-valued argument, t, as in Equation 2.1.

$$s(t) = \int x(a)w(t-a)da \tag{2.1}$$

The operation is typically denoted $s(t) = (x{*}w)(t)$. While the definition in Equation 2.1 is for arguments in the continuous range, data in computer applications is usually discrete. If we assume that x and w are only defined on discrete values of t, the convolution operation can be discretized, as defined in Equation 2.2.

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.2}$$

In CNN terminology, x and w represents the input and the convolution kernel, respectively. For a network, the input and kernel are usually multidimensional arrays, the first containing data and the second containing the network parameters, or weights. Because these arrays are finite, we can assume that the functions are zero for elements that reside outside the arrays' range. The infinite summation in the discrete convolution operation can then be reduced to a summation over a finite number of array elements. In addition, it is often desirable to perform convolutions over multiple axes at a time. The convolution operation for a 2D image input I, using a 2D kernel K, would then be defined as in Equation 2.3.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.3}$$

Since convolution is commutative, the definition in Equation 2.4 is equivalent.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.4}$$

The commutative property is a result of the kernel being flipped relative to the input, such that the input index increases with m, while the kernel index decreases. This property is not essential in practice and many ANN libraries implement cross-correlation instead, calling it convolution. In reality, cross-correlation is equivalent to convolution without the kernel flipping, shown in Equation 2.5.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.5}$$

An example of an application of cross-correlation on 2D input can be seen in Figure 2.4 Following convention, we will not differentiate between the two approaches any further, and refer to both as convolution operations.

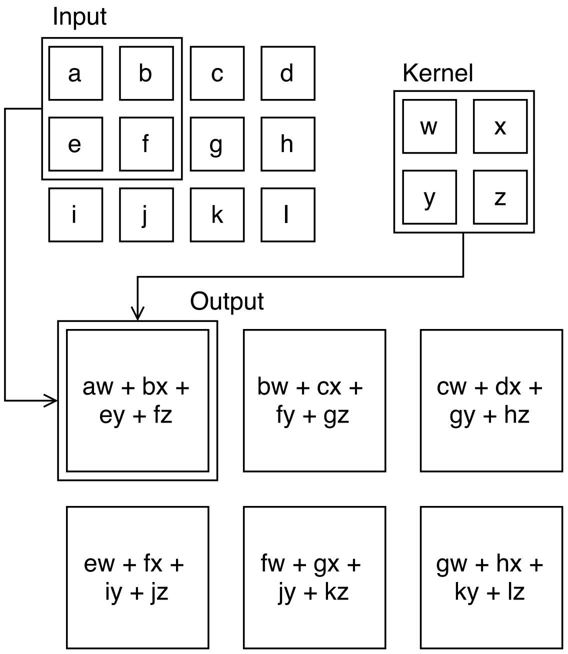


Figure 2.4: An example of 2D cross-correlation. Arrows indicate how marked input and kernel combine to create output. The example uses single strides and no padding. Figure found in Goodfellow et al. [16].

2.2.2 Convolution as Matrix Multiplication

Discrete convolution can be represented by a matrix multiplication and there are multiple ways to define the operation. The definition presented here was chosen as it allows the differences between convolution and transposed convolution to be easily demonstrated in Section 2.2.6.

Consider a discrete convolution of a 4x4 input matrix with a 3x3 kernel matrix. To perform this convolution via matrix multiplication, we must first unroll the input matrix from left to right and top to bottom, reshaping the input into a 16-dimensional vector I. The kernel must also be modified, creating a sparse matrix K, where w_{ij} is the element at row i and column j in the original kernel, as seen in Equation 2.6 [17].

$$K = \begin{pmatrix} w_{00} & w_{01} & w_{02} & 0 & w_{10} & w_{11} & w_{12} & 0 & w_{20} & w_{21} & w_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{00} & w_{01} & w_{02} & 0 & w_{10} & w_{11} & w_{12} & 0 & w_{20} & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{00} & w_{01} & w_{02} & 0 & w_{10} & w_{11} & w_{12} & 0 & w_{20} & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{00} & w_{01} & w_{02} & 0 & w_{10} & w_{11} & w_{12} & 0 & w_{20} & w_{21} & w_{22} \end{pmatrix} \tag{2.6}$$

Executing the matrix multiplication KI would then produce a 4-dimensional vector, which could be reshaped into the 2x2 matrix we expect from the convolution. Thus, the matrix multiplication is equivalent to the regular convolution operation.

2.2.3 Convolutional Layer

To reiterate, a convolutional layer employs convolution to compute its output, unlike traditional fully connected layers. As a result, convolutional layers incorporates certain advantageous concepts, namely sparse connectivity, parameter sharing and equivariance to translation.

Sparse Connectivity

As implied by the name, every unit in a fully connected layer is connected to every unit in the previous layer. In convolutional layers, however, the convolution kernel is typically smaller than the input, causing sparse connectivity. Each unit in the layer is then only connected to a subset of the units in the previous layers, as dictated by the kernel. Consequently, the output of each unit can be computed using fewer mathematical operations. Sparse connectivity also results in a lower amount of weights compared to a fully connected layer, and even drastically so when combined with the effects of parameter sharing.

Parameter Sharing

Where each weight of a fully connected layer is only used once, each weight in a convolutional layer is used multiple times. When convolving, the kernel is applied at several different areas of the input, reusing the parameters of the kernel across the input. Every unit in the convolutional layer then uses the same kernel weights as the other units to connect to the layer below. In this situation, the weights are said to be shared among the units. Additionally, the parameter sharing characteristic means that the kernel parameters learned in training are location independent.

Equivariance to Translation

The equivariance property of convolutional layers is caused by the particular form of parameter sharing. Specifically, convolutional layers have equivariance to translation, which means that if the input of the layer experiences any translation changes, then the output changes in the same way. Consider a convolution with an image input, where the kernel parameter values are set to detect the edges of an object. The convolutional layer would then create a map of where the edges are located. Because of the equivariance property, if the object shifts to another location in the image, the edge mapping would shift accordingly.

Convolution for Feature Detection

Together, the discussed concepts make convolutional layers a powerful addition to the ANN toolbox, especially for image input. As mentioned, the kernel parameters can be chosen so that the convolutional layer detects edges. Since the same parameters are used across the input, the layer will be able to discover edges at any location in the image. Edges might not be the only thing of interest, and using a kernel with different parameters could provide a mapping of another feature, for example a corner. Consequently, convolutional layers often employ multiple different kernels, which differ in parameter values, but are equal in size. In truth, detecting edges with a single kernel would only allow the layer to identify edges of a specific orientation, for example horizontal of vertical. With several kernels, the layer can detect edges of any orientation, as well as many other simple features.

A second convolutional layer could be added, convolving the output of the previous layer with its own specialized kernels. This enables the detection of features that are composed of the features found in the preceding layer. With additional layers, the features can become even more complex, each building on the ones below. Stacking convolutional layers make CNNs able to perform remarkably well on image input, for example in face recognition. Then, lower layer kernels start by detecting simple features, like edges, which later kernels can use to detect more complex patterns, like eyes or mouths. Layers even deeper in the network can then use that output to identify complex structures like faces. While the sparse connectivity between layers might be thought to hinder such complexity, the connectivity of units in later layers take on a pyramid-shaped pattern, as throughout the layers they become indirectly connected to an increasingly larger portion of the units below and, ultimately, the input. Because of the feature detecting abilities of convolutional layers, their kernels and outputs are often referred to as filters and feature maps, respectively.

When convolving with several kernels, the output of a convolutional network gains a new dimension. On a 2D image, a 2D kernel will produce a 2D output. With several kernels, the output assumes a 3D shape, with the kernel number following the depth axis. In reality, the image input is also usually 3-dimensional, on account of the color axis for RGB images. Thus, convolutional layers often operate with 3D input, either the original image or the feature map of the layer below. In these situations, kernels are also 3-dimensional, with sparse connectivity in the spatial dimensions, but full along the channel (color/kernel) axis. Used on feature maps, these kernels can then consider all the different features identified simultaneously, enabling the complex feature detection. The output will be 3-dimensional, as before. If the next layer is fully connected, it is common to flatten the output.

2.2.4 Hyperparameters in a Convolutional Layer

The effects of the hyperparameters described here are mostly structural and affects the single convolutional layer for which they are used. In a CNN, however, the hyperparameters of one layer can have widespread consequences, which depend upon the domain and network architecture. Instead of an exhaustive explanation of the potential ramifications, we present the concepts and more immediate effects. There are five important hyperparameters to consider: the activation function, amount of filters, filter size, strides, and padding.

Activation Function

The activation function in a convolutional layer is employed similarly as in a fully connected one, computing the output of every unit in the layer. It is typically chosen to be a ReLU function, to add non-linearity to the network and ensure strictly positive feature maps. The latter is useful, as a negative activation for feature detection is not meaningful.

Filter Amount and Size

Not surprisingly, the filter amount denotes how many different filters the layer should use and the filter size determines the spatial size of the filters, which is usually small, e.g. 3x3. As previously mentioned, a convolutional layer can use multiple filters to search for different features. Intuitively, the amount of filters a layer uses dictates how many different features it can extract. In addition, it also decides the depth of the layer output. The filter size affects the output size as well, which is explained in more detail when reviewing padding.

Strides

The stride refers to the distance between filter applications in units. For image input, it applies in both spatial directions, meaning two separate strides must be specified. In a basic convolution, with single strides and no padding, the filter would be applied to every unique (spatial) neighborhood of units that matches the filter size. For a 4x3 input and a 2x2 filter, there exist six such neighborhoods, as implied by Figure 2.4. Each filter region is then a single step away from any another, both vertically and horizontally. If we increase the horizontal stride to two, the filter will only be applied to every second neighborhood in that direction. As a result, four regions of the input would be convolved, creating a 2x2 feature map. By increasing both strides to two, the feature map would be further reduced to 2x1, as there would not be enough units for two filter regions that are two steps apart in the vertical direction. The strides can be further increased, but they are rarely set to large values [1]. The purpose of the strides is to downsample the input, reducing computational cost at the expense of finer feature extraction. Although increasing the filter size would also diminish the feature map, it would limit the layer's feature detection capabilities, making larger strides the preferred approach.

Padding

Padding has the opposite effect of strides and is used to either retain the spatial size of the input or increase it. Even without strides, any input passed through a convolutional layer will decrease in height and width by one less than the filter height and width, respectively. This happens because the layer will only apply convolution where the filter fits and not at the border of the input, where parts of the filter fall outside. If the layers keeps diminishing the input, it puts a restriction on the amount of convolutional layers that a network can employ before the output becomes too small for any meaningful use. An alternative is to use smaller filters to reduce shrinkage, but this limits the expressive power of the network similar to the layer restriction. However, by zero padding the input before applying filters to it, the problem can be avoided.

When padding is involved, convolution can be separated into three categories: valid, same and full. Valid convolution occurs in the aforementioned case, with no padding, where convolution is only applied at regions that fully contain the filter. In same convolution, padding is added to the layer input to make the layer output have the same spatial size as the unpadded input. However, the units at the border of the input will still be involved in less convolutions and therefore influence less layer units. This is taken into account in full convolution. Here, padding is added so that every input unit is involved in an equal amount of convolutions. Some layer units will then be a function of fewer input units, which can interfere with the learning of decent filters. Consequently, valid and same convolution are usually preferred over the full variation.

2.2.5 Training a Convolutional Layer

Training a convolutional layer is similar to training a fully connected one, as explained in Section 2.1.4. In Section 2.2.2, convolution is defined as a matrix multiplication. Following the example from that section, we use the kernel matrix K from Equation 2.6 to define the backward pass for convolutional layers. While we have established that convolutional layers often handle input, output and kernels with three dimensions, we use 2D matrices here to simply illustrate the backward pass approach. To that effect, we also assume single strides and no padding.

The Backward Pass

The forward pass has already been established to be the matrix multiplication KI, using the 16-dimensional input vector I, which produces a 4-dimensional output vector. To do a backward pass, the transposed version of K, K^T, would be used. This transposed version of the convolution matrix allows the backward pass to receive a 4-dimensional vector as input and produce a 16-dimensional vector as output, which can be reshaped to the original 4x4 size. If the error is being backpropagated, the backward pass would use a loss vector, L, to compute $K^T L$.

Connectivity Pattern Conservation

In a forward pass, a single entry in the output vector is only affected by a subset of the entries in the input vector and always the same ones. The backward pass has the same connectivity pattern between the entries in the input and output vector, only in the reverse order. These connectivity patterns are compatible since both passes are defined by K. In context of the example, the first entry (index 0) of the forward pass output vector would be connected to the input vector entries with index 0-2, 4-6, and 8-10, as implied by the first row of K. Subsequently, the backward pass output vector entries with index 0-2, 4-6, and 8-10 would be connected to the first entry in the input vector, as implied by the first column of K^T. The conservation of this pattern affirms the validity of the backward pass approach.

Effects of Strides and Padding

Strides and padding have an effect on the output size of a layer, as previously explained. These two, in combination with input size and filter size, determine the spatial size of the feature maps. Using different stride and padding policies, forward passes for multiple inputs of varying size can produce feature maps of similar size. As a result, performing a backward pass transposed convolution will not necessarily be as straightforward as in the basic case, with single strides and no padding. The details on how transposed convolution can be performed with the various combinations of strides and padding is not of great importance to the thesis and will be described no further. It suffices to be aware of the issue, which explain why some implementations require the desired output size to perform transposed convolution.

2.2.6 Transposed Convolutional Layer

A transposed convolutional layer (also known as a fractionally strided convolutional layer) is almost identical to a regular convolutional layer, with an important distinction: the forward and backward passes are swapped. Where a regular layer uses K and K^T for the forward and backward pass, respectively, the transposed convolutional layer would use K^T and K, respectively. As $(K^T)^T = K$, the transposed convolutional layer is simply a regular convolutional layer using transposed convolution matrices, hence the name. Because both the passes are defined by K, the connectivity pattern found in regular layers also holds for transposed convolutional layers.

Transposed convolutional layers can be implemented with the same hyperparameters as regular convolutional layers. In this case, the strides and the padding specified usually applies to the convolution operation in the backward pass, which then dictates how the transposed convolution in the forward pass will behave. These layers are typically found in networks used to reconstruct some input from the output of another CNN. An example is deconvolutional neural networks, which try to reconstruct an input image, with certain features highlighted, from a set of feature maps. Deconvolutional networks will be explored further in Section 2.3.4.

2.2.7 Max Pooling Layer

Pooling layers are a common occurrence in CNNs and they can employ multiple types of pooling functions. A pooling function is applied to various regions of the input, and outputs a summary statistic of the units for each region. Popular ones include max pooling, average pooling, and weighted average based on distance from pooling center. Only max pooling is relevant for the thesis and will be described in this section.

Max pooling layers are fairly straightforward. Given an input, a layer determines several pooling regions and outputs the maximum value for each region. Pooling regions are similar to convolutional filter regions, with the exception that they are usually singular in depth. The region spatial size is decided by a pooling size hyperparameter. Pooling is also typically restricted to full regions, e.g. a 3x3 pooling will not be applied to a 3x2 region at the border of the input. Like convolutional layers, however, pooling layers can employ strides and padding to influence how input is processed.

Translation Invariance

The benefit of pooling layers is that they make a network approximately invariant to small translations of the input. Remember that convolutional layers have equivariance to translations, so their output shifts similarly to the input. In pooling layers, a small shift of the input will only affect the output of the regions that gain a new maximum value. This only happens in regions where the previous max unit is shifted out or if a unit with greater value is shifted in. In practice, most of the output will remain unchanged, providing approximate invariance. It is a useful property to have if it is important to detect whether a feature is present, but its exact location is less significant. Some times, the location of a feature is important, for example if we are looking for a corner defined by the connection of two edges, the edge locations must be preserved properly to examine whether they join or not.

Training

CNNs with max pooling layers can still be trained with gradient descent and backpropagation. There does not exist an inverse to the max pooling function to use in a backward pass, but that problem can be solved using so-called switches, which are routings between input and output units. In a forward pass, simply save which input units had a maximum value and which output unit they correspond to. These routings can then be used in backpropagation to pass the error back to the right unit.

2.3 Visualization

When using ANNs to solve complex problems, the inner workings can be difficult to understand. It is not unlikely that the network is highly non-linear and requires millions of parameter values to obtain a desirable performance level. In these cases, it is hard to gain insight about the network's behaviour by examining the parameter values directly, both because of the unfeasible amount of values and each value's relative insignificance by itself. However, when applying ANNs to tasks in the image domain, there are several ways to help us understand the inner workings of the network by presenting information in the same readable format as the input. Aside from the fact that images are more suited than numbers for human readability, there exist several techniques that utilizes the connections in the network to produce visualizations. These images can then provide insight into a larger part of the network, spanning several layers, possibly the whole network. Through the use of these visualization techniques, the network can be improved by making informed decisions, instead of employing a trial and error approach.

2.3.1 Training Progress Measurements

While training, the progress of the network can be visualized in a simple line plot. Measurements could be made of the standard network appraisal values, loss and accuracy, both on the training set data and, if provided, the validation set data. A typical thing to want to monitor, these values provide a means to explicitly evaluate a network's performance. Plotting the values against time, at certain batch intervals or at epoch completion, puts the current measurements in perspective, and makes the training progress, or lack thereof, easily discernible. The shape of the line plot could also be used to reveal undesirable values in the training hyperparameters. For example, if the learning rate is too large, the network is likely to vastly improve at first, only to have its progress quickly stagnate. In this case, both loss and accuracy plots would have distinct shapes that indicate a problem, without having to look at any of their actual values. By including the validation values, one could also be able to identify the presence of overfitting, marked by a point where the validation accuracy begins to decline while training accuracy continues to rise.

2.3.2 Layer Activations

Visualizing the activations of the layers in a network is a straightforward, yet useful visualization technique. It is informative because all data flowing through the network can be visualized [18]. The activation output is most readable for the convolutional layers, and especially so for the earlier ones. For a convolutional layer, the activations of each filter of the layer are typically displayed in a grid of feature maps. The feature maps denote high activations with a bright color, and low activations with a dark color. The brighter the color, the higher the activation.

The layer activations can be used to determine what features a network learns to detect at its various layers, by examining the similarities between the areas where the filter activates. The activations can also be helpful to confirm that the network is progressing well, if the output is showing signs of activation in areas with similar features. These patterns should become more and more clear as training goes on, and when the activations start experiencing little or no change, the proper filters should have been obtained. In networks using ReLUs, a common activation function, the expectation of seeing sparse and localized activations can be confirmed, which is an indicator of the network learning properly. If some feature maps are entirely dark for many different input images, this can be a warning sign that the learning rate is too high. The corresponding filters are then rendered useless, and they have no possible way of recovering [19].

2.3.3 Saliency Maps

A simple, yet useful, visualization technique for ANNs is to compute saliency maps [20]. Saliency is defined to be ”a pronounced feature or part”, i.e. something that makes an item stand out. For images, a saliency map is thus a visualization that displays each pixel's saliency, or its unique quality. The notion of saliency is commonly used as a part of image segmentation. In relation to an ANN, however, this unique quality would be a specific pixel's influence on the output. Applying this to a classification problem, the output would be a specific class that the ANN believes that the input belongs to. A saliency map could then be computed for an input image and a class, to indicate to what degree each pixel in the input image influenced the specific class outcome. The produced image is typically black and white, with insignificant pixels having a darker color, and important pixels having a lighter color. Thus, the important regions can be easily identified by their brightness. In other words, we can view which parts of an image that the network deems more important when deciding upon the classification score chosen.

A saliency map would typically be computed for an input image and its resulting top classification. To obtain the saliency map, backpropagation must be employed to find the gradient of the loss of the selected class with respect to the image. This gradient will be a matrix with the same dimensions as the input image. The absolute value of the gradient matrix should then be calculated and the entries scaled to match standard pixel values. If the input is an RGB image, the matrix should be transformed from 3D to 2D by computing the maximum values across the color channels. The resulting matrix corresponds to a grayscale image that represents the saliency map of the input for the chosen class. The map and input have equal size in the spatial dimensions, and the map entry at *(i,j)* shows the saliency of the input pixel at *(i,j)*, RGB or otherwise. Note that it is also possible to compute the positive saliency, i.e. a map that shows how the pixels positively correlate with the output, by discarding the negative gradient values instead of computing their absolute. The negative saliency map can be acquired in a similar manner by discarding the positive gradient values.

2.3.4 Deconvolutional Network

This section is based on the work of Zeiler and Fergus [21], unless otherwise specified.

Modern ANNs that receive images as inputs are likely to have convolutional layers form the initial part of the network. This part is aimed at detecting certain features in the image, with the complexity of the features rising as the depth of the convolutional part increases. In such cases, we can utilize a visualization technique that employs a deconvolutional network, which can be thought of as a reverse convolutional network [22]. The visualizations are used to interpret the feature activity in the intermediate layers of the original network. To obtain a visualization, an image is first passed to the original network to produce activations in its various layers. The activations of the convolutional part can then be processed and passed to the deconvolutional network to create a visualization of the pattern in the input image that was responsible for eliciting the given activations. This pattern can then reveal what a certain part of the original network finds important and what it is looking for in an image.

Constructing a Deconvolutional Network

To build such a deconvolutional network, it is necessary to examine the original network we want to produce visualizations for. Only the feature extracting part is of interest, and we can regard this part as a network on its own. The purpose of the deconvolutional network is to map the feature activity present in this convolutional network back to the input pixel space. For a convolutional network consisting of convolution and max pooling layers, the corresponding deconvolutional network would consist of transposed convolution and unpooling layers in the opposite order of the original layers. In addition, where a ReLU activation function commonly follow each convolutional layer, they would precede transposed convolutional layers. In other words, a top-down examination the convolutional network would enable a bottom-up creation of the deconvolutional network.

Mapping Features to Input

The fact that a transposed convolution can reverse a convolution downsampling while maintaining a similar connectivity pattern makes them suitable for a deconvolutional network. A convolutional layer in the original network would apply their filters to the output of the preceding layer (or the image input) and produce feature maps, which are then rectified by the ReLU activation function. In the deconvolutional network, the aim is to map the rectified feature maps back to convolution input, and this is where transposed convolution is used. By using the same filter weights in a transposed convolutional layer as in its corresponding convolutional layer, the deconvolutional network is able to create a mapping that is specific to the original network.

As previously mentioned, the input to the transposed convolution should first be passed through a ReLU activation function. In a convolutional layer, ReLU ensures that the feature maps outputted are strictly positive. For the convolution input reconstruction to be valid, the reconstructed feature maps used as input for the transposed convolutional layers must also be strictly positive. Passing the reconstructed feature maps through a ReLU activation function ensures this.

Approximating Unpooling

To reverse the effects of the max pooling layers of the original network, the deconvolutional network uses unpooling layers. Pooling, however, is not an invertible operation and the unpooling layers therefore focus on obtaining an approximation of the original max pooling input. The key to the approximation is to use a switch variable for each pooling region. When performing max pooling, these switches record the location of the chosen maximum in each region and pass the information to the unpooling layer. Here, they are used to place the entries in the unpooling input into the appropriate locations in the output. The result is an unpooling output that only has non-zero values in entries that correspond to the maximum entries in the pooling input. This reconstruction approach preserves the structure of how max pooling processes stimulus.

Using a Deconvolutional Network

To utilize a constructed deconvolutional network, the feature map output from a chosen layer in the original network must first be produced. Before the feature maps are passed to the corresponding layer of the deconvolutional network, they must be processed. To visualize what a particular filter is looking for, all entries in the feature maps are set to zero, except the maximum entry in the filter's corresponding feature map. Only the maximum entry is left untouched to create a visualization that is able to convey what pattern in the input image activates that filter strongly. When the processed feature maps are inputted into the deconvolutional network, the layers will successively reconstruct the activity that was responsible for the chosen feature map activation until the input pixel space is reached. If visualizations for several filters are desired, the deconvolutional network has to run multiple times, each with feature maps only containing a single non-zero entry.

Visualization Output

When the original network is applied to an input image, the switches created in its max pooling layers are specific for that image. The reconstruction from the processed feature maps are created using these switches and it will therefore resemble the input image. However, only the patterns that contribute towards the chosen feature map activation are reconstructed. The visualization outputted by the deconvolutional network will then be mostly blank, with a highlight of the important section of the image and its characteristics. Creating visualizations from several images for a single filter can reveal what the filter is looking for. For example, if a filter is trained to detect eyes, the reconstructions from this filter will repeatedly focus on image sections containing eye-like traits, as well as highlighting those traits. One could also have a single image and use several different filters to produce visualizations, which could reveal if the original network is capable exploiting the discriminative features in the image. If a network is supposed to examine faces, but none of the visualizations show any focus on the eyes, they could warn about flaws in the network and potentially explain poor performance. Conversely, if the visualizations cover a wide range of facial characteristics, they can provide reassurance about the quality of the network's feature utilization.

Visualizing for several layers can also shed light on the hierarchy of the filters, where the lower layers usually concentrates on low level features like edges and textures and the upper layers looks for more complex features, e.g. eyes. If a network experiences problems with upper layer filters, it is possible that these stem from poor filters further down. Applied during training, visualizations from different layers reveal when the filters of each layer converges. Typically, the lower layer filters converge quickly, while the upper layer filters develop at a much slower rate.

2.3.5 Deep Visualization

The deep visualization theory presented here is based on the paper from Yosinski et al. [18], unless otherwise specified.

Whereas the previously discussed techniques (aside from plotting) are all dependent on a provided image to stimulate the network in order to create visualizations, deep visualization is not. Instead of examining the network's reaction to certain input, deep visualization aims to create images that represent what a selected part of the network is looking for. In order to achieve this, several steps are taken.

At first, an individual unit in the network must be chosen. It is this particular unit that the visualization will offer insight into. Secondly, an initial image should be created by setting every pixel to a random value. The dimensions should match the network's expected input. The next step is to compute an update to the image using gradient ascent. By iteratively updating the pixel values with the gradient of the loss of the chosen unit with respect to the image, the technique is able to produce images that maximally activate the unit. The idea is that images that cause that strongly activate the unit would resemble the object or type of input that the unit looks for, i.e. its learned features. However, images produced from gradient ascent alone are inclined to contain certain hacks to influence the activation. These hacks, like extreme pixel values and structured high frequency patterns, guide the process away from natural, interpretable images. Fortunately, this behaviour can be suppressed. By employing regularization techniques to deter the evolution of hacks in the image, deep visualization will produce visualizations that are able to reveal the learned features of the unit. By repeating these two steps, updating the image with the gradient and then regularizing it, the process will eventually result in a viable visualization. When updating the image with the gradient, there can also be a learning rate attached, to accelerate (or decelerate) learning. It is worth noting that the process is stochastic as a result of the random initial images. Producing several visualizations for a single unit would therefore display certain variances. These variances in turn reveal the invariances learned by the unit.

As an example, consider a network trained on classifying certain objects from images. In its last layer, this network has a unit which activates to classify the object in the input image as a cat. If this unit is selected for deep visualization, the resulting images should display various catlike features, e.g. four legs, tail, furry texture, particularly shaped faces, etc. If the output unit visualizations contain the expected features, confidence in the network is reinforced. However, if the features portrayed are irrational or intangible, it signals potential issues with the network. Assuming the network is able to classify a cat, it most likely contains parts that look for eyes as well. As deep visualization can produce images for any units in any layer, a hidden unit which activates on eyes could also be visualized. In that case, eyelike features should gradually become discernable in the once random image. Visualizing for a wide selection of hidden units can be helpful to understand the scope of the network's feature learning abilities. Ultimately, deep visualization can aid in understanding the network and whether it is looking for sensible features.

Regularization is the key to producing interpretable visualizations and the techniques used are L_2 decay, Gaussian blur and clipping based on specific attributes. While each is somewhat beneficial alone, they are even more effective when used in an ensemble. Before examining some suggested hyperparameters for the various regularization techniques, each technique is explained more thoroughly.

L_2 Decay

Employing L_2 decay is a common regularization technique for ANNs, where it is used to penalize large weight values. In these cases, the weights are linearly decayed towards zero with a chosen regularization strength [23]. In deep visualization, L_2 decay is similarly applied to the visualization, where it is used to penalize extreme pixel values and prevents them from dominating the image. This is desirable as extreme pixel values are neither naturally prevalent nor useful for visualizations.

Gaussian Blur

Without regularization, an image generated from deep visualization has a tendency to contain a great deal of high frequency information, i.e. the pixel values are often radically different from that of their neighbours. While natural images do contain some high frequency information, for example at edges of objects, an excessive amount of it leads to unrealistic and uninterpretable visualizations. To combat this problem, Gaussian blur is applied as a regularization technique. The blurring makes transitions between pixels smoother, lowering the amount of high frequency information.

Of the techniques implemented, Gaussian blurring is the most computationally expensive. To reduce the cost, blurring can be performed at certain intervals, using a blur kernel with a greater standard deviation. A single application of blurring with a large standard deviation kernel is equivalent to blurring multiple times with a small standard deviation kernel. Although the image will change slightly in between applications, the desired effect will be similar.

Clipping Based on Value, Norm and Contribution

Another characteristic of images produced using deep visualization is an abundance of non-zero pixel values. In general, natural images contains a myriad of information aside from the primary focus. While not necessarily essential to the network, the secondary information will influence it nonetheless. In the visualizations, it manifests as disruptive non-zero pixel values. A subset of pixels will show the main object or type of input that excite the chosen unit and these are the focal point of the visualization technique. The other pixels, however, will still generally influence the gradient, each with a minor contribution to raise the activation. As a result, unwanted patterns will emerge in the visualization, disturbing the primary focus and cluttering the image. To prevent this behaviour, these disruptive pixels should be set to zero, which can be handled by using clipping regularization based on different attributes, namely value, norm and contribution.

Clipping based on value is rather straightforward. If the absolute value of a pixel in the visualization is below a set threshold, set the pixel value to zero. For an RGB image, each color value can be examined separately. Although using values for clipping is not very complicated, it is not especially sophisticated either. Another approach is to clip pixels based on their norm. The norm of each pixel would be computed over their color channels, and each pixel with a norm under the chosen limit would be set to zero. Finally, the pixel contributions to the unit activation can be used. The pixel contributions could be found by measuring how activation changes when each pixel is set to zero, but it would require a forward pass for every pixel and therefore be excessively expensive. Instead, the pixel contributions can be approximated by the elementwise product of the visualization and the gradient. For an RGB image, each pixel contribution would be the sum of its color channel contributions. Normally, a pixel would be clipped, i.e. set to zero, if the absolute value of its contribution is below a set threshold. This would discard pixels which have a small contribution in either a positive or negative direction. Alternatively, one could use the contributions without computing their absolute values. In this case, pixels which greatly affect the activation in a negative manner are more likely to fall beneath the threshold and be clipped. The arguments against this is that the approximation made limits the validity of such large shifts and that the gradient ascent will handle these pixels iteratively. Thus, it could prove better to restrict regularization to only clip pixels that are considered to have an insignificant contribution value.

Regularization Hyperparameter Values

Through a random hyperparameter search, Yosinski et al. [18] found several useful combinations that create visualizations with certain desirable characteristics. Not all of the previously described regularization techniques are included, however. Both clipping based on value and clipping based on contribution are included in the web article of Yosinski et al.[1], but not in the Yosinski et al. paper [18] (the latter is briefly mentioned). As a result, they are not included in the hyperparameter suggestions, but have been explained for the sake of diversity.

Denoting L_2 decay rate as θ_{decay}, blur interval as $\theta_{b\_interval}$, blur kernel standard deviation as θ_{b_std}, norm percentile threshold as θ_{norm} and absolute contribution percentile threshold as θ_{abs_contr}, the hyperparameter combinations are shown in Table 2.1.

θ_{decay}	$\theta_{b\_interval}$	θ_{b_std}	θ_{norm}	θ_{abs_contr}
0	4	0.5	50	0
0.3	0	0	20	0
0.0001	4	1.0	0	0
0	4	0.5	0	90

Table 2.1: Hyperparameters found in random search, with useful combinations located in rows. The four combinations produce visualizations that complement each other. The first one can be used to produce images that show a sparse set of important regions. The second tends to result in images with high frequency patterns. The third favors images with low frequency patterns. Finally, the fourth one generates visualizations with dense pixel data. Used separately, the most beneficial images can be developed from the third combination, although more insight can be gained by examining visualizations from all four combinations at once.

2.4 Transfer Learning

The idea behind transfer learning is that knowledge learned from solving one problem can be used to solve a different, but related problem. Specifically related to image recognition, research has shown that CNNs trained on images learn generic features for the first couple of layers, and that these features are applicable for several different datasets and tasks [24]. Modern CNNs are usually very deep and requires a signification amount of time to train, and therefore many researchers choose to publish their fully trained networks, including weights, allowing others to use them for their own purposes. Using transfer learning, these networks can be employed in a wide variety of tasks, with only a minor amount of modification and training. There are two primary approaches for employing transfer learning: using the CNN as a fixed feature extractor, and fine-tuning the CNN.

[1]http://yosinski.com/deepvis

2.4.1 Fixed Feature Extractor

The first strategy is to use a pretrained CNN as a fixed feature extractor for a new dataset. For classification, this is accomplished by replacing the last layer of the network with a new fully-connected output layer with a size matching the new data. The CNN should then be trained on the new training data, keeping the weights fixed in all layers, with the exception of the new output layer. Typically, a lower learning rate is used compared to training a full network. This approach works well if the new dataset is small and similar to the training set that the pretrained CNN has been trained on. If the new dataset is significantly different, it could be beneficial to replace more than just the last layer, since the top layers of a CNN detects more specific features than lower ones [25].

2.4.2 Fine-Tuning

Fine-tuning a pretrained CNN includes not only replacing the top layer of the network, but to also fine-tuning the weights by continuing the backpropagation through the network. The fine-tuning can be performed for the whole network, or only the top part by keeping the lower layers fixed. It is important to use a low learning rate when fine-tuning. Otherwise, the error backpropogated from the newly added layers will result in large and disruptive gradient updates in the pretrained layers, undoing their feature extraction capabilities. If the new dataset is large, fine-tuning can be a good option as overfitting will not likely be an issue.

2.5 Facial Expressions

Face recognition in unconstrained environments remains difficult for several reasons. Among the challenges are facial expressions, pose, illumination, aging effects, and poor image quality. A facial expression can be described as the combination of a number of variations in a person's facial features. These variations are caused by the contraction or extension of facial muscles, and effort has been made to categorize the effects of muscle activity on facial features into a Facial Action Coding System (FACS) [26]. FACS encodes facial displays into action units, which specify the muscles involved and how they affect the facial expression. The system is widely employed in expression research and has, among other things, been used to create a facial expression database [27], to further fuel the progress of facial expression analysis. The intricacy and extent of human emotional expressions have also motivated attempts at creating a confined and discrete set of human facial expressions. A popular such set is Ekman's six basic expressions: happiness, surprise, sadness, anger, disgust and fear [28].

2.5.1 CNNs for Expression Recognition

There exists a long history of research into analyzing facial expressions in human subjects. For computer systems, researchers have been interested in creating automatic expression analysis, which is comprised of three steps: face acquisition, facial data extraction and representation, and facial expression recognition. Extracting and representing facial data can either be completed using geometric feature-based methods or appearance-based methods. The former present the shape and location of certain facial components, like the mouth or eyes, in a vector to represent the face geometry. The latter utilizes filters to extract feature vectors, either applying them to the full image of a face or at specific regions of it [29].

Face expression recognition is a task that CNNs have been successfully applied to. A CNN for expression recognition incorporates all of the aforementioned three steps in single network, using an appearance-based method with convolutional filters. Examples are the Action Unit-inspired Deep Network [30] and DeXpression [31]. Both network produce state of the art competitive results for expression recognition performance on the Extended Cohn-Kanade database [32]. The former achieves 93.70% accuracy, while the latter achieves 99.60% accuracy. These results demonstrate that CNNs have the ability to extract and recognize expressions from image input.

2.5.2 Overcoming Expressions in Face Recognition

To construct face recognition systems that are invariant to facial expressions, some procedures try to explicitly exploit the changes made to face geometry by expressions [33, 34, 35]. With state of the art ANNs, however, it is common to overcome the challenges of expressions by training on large, varied datasets [36, 37, 38]. By exposing the networks to images with a wide variety of expressions, they learn how a face can have variations in its features, but still represent the same identity. The face representations learned by such training are robust [39], which helps explain how CNNs can achieve high performance without directly addressing issues like facial expressions, pose, and illumination.

Chapter 3

Related Work

This chapter presents a selection of the work related to our research, in terms of existing visualization tools and face recognition systems.

3.1 Visualization Tool

This section will present existing visualization tools for ANNs. The advantageous features of each tool will be described, as well as their limitations, before ending the section with a discussion of the improvements offered by our approach.

3.1.1 Existing Visualization Tools

We describe three prominent visualization applications for ANNs, namely TensorBoard[1], Deep Learning GPU Training System (DIGITS)[2], and the Deep Visualization Toolbox[3]. While all are functional, the former two are being developed further, both improving existing features and adding new ones. Both of these are backed by large companies, with Google supporting the open-source project TensorFlow, and by extension TensorBoard, and NVIDIA developing DIGITS[4]. The Deep Visualization Toolbox was developed as part of a research paper and is not receiving any support or updates [18]. Other visualizations tools found had either too narrow a scope or too rudimentary an implementation to be considered. While the three tools differ greatly, their main purpose coincide: to aid in the creation and understanding of an ANN.

[1]https://www.tensorflow.org/get_started/summaries_and_tensorboard
[2]https://github.com/NVIDIA/DIGITS
[3]http://yosinski.com/deepvis
[4]https://developer.nvidia.com/digits

TensorBoard

TensorBoard is a suite of visualization tools for networks built with the TensorFlow library. The purpose of TensorBoard is to make it easier to understand, debug, and optimize ANNs. It can be used to visualize TensorFlow graphs, which is how TensorFlow networks represent computations, providing a user with an architectural and computational overview. TensorBoard can also be used to plot network metrics over time, such as how the learning rate decreases or the loss changes during training, and other statistics, like the distribution of weight values. The user activates this by annotating the nodes of the graph with so-called summary operations, which prompts TensorBoard to write network summary data to a log and then produce real-time visualizations of its content. TensorBoard can also visualize high-dimensional data, like embeddings, through an interactive user interface.

DIGITS

The NVIDIA Deep Learning GPU Training System, also known as DIGITS, is a web application developed for managing deep ANNs with input in the image domain. Originally, it only supported networks created with the Caffe[5] library. A drawback with Caffe is that the network architecture needs to be defined in plain text configuration files, which can be tedious, especially for larger networks. DIGITS simplifies this process by providing users with an intuitive interface for executing many of the cumbersome tasks that usually requires manipulating the configuration files directly. At a later point, the application expanded its support to include the scientific computing framework Torch[6]. The application allows users to upload and manage both datasets and network models, and also makes pre-trained models, such as AlexNet [40] and GoogLeNet [41], available for use. For their uploaded networks, users can schedule, monitor, and manage training jobs, and view real-time visualizations of the accuracy and loss. In addition, DIGITS is able to execute networks on uploaded input, providing an output result, and simple visualizations of the weights and activations of each layer.

As mentioned, DIGITS is still being developed and its has been upgraded since the start of our Specialization Project. The developers have previously stated that they had no plans towards adding support for more frameworks, including TensorFlow[7]. However, in Q2 2017, the DIGITS website[8] was updated with a message stating that TensorFlow support would be available in July 2017.

Deep Visualization Toolbox

The Deep Visualization Toolbox is a software tool that provides an interactive visualization of a trained ANN as it responds to user-provided input. The input can either be an uploaded image or video from a live web camera feed. Users can choose to use the included default

[5]http://caffe.berkeleyvision.org/
[6]http://torch.ch/
[7]https://github.com/NVIDIA/DIGITS/issues/967
[8]https://github.com/NVIDIA/DIGITS

network or opt for a network of their own making. The latter requires additional computation to produce the more complex visualizations in advance. The toolbox only supports networks created with Caffe. The toolbox allows users to cycle through layers and view the visualizations made for each one. These are either the activation output of the layer or a synthesized image that causes high activations in the layer units. In addition, a specific filter can be selected to be explored further. For a chosen filter, the toolbox can show nine synthesized images that induces activation, the nine images from the training set that activates the filter the most, and the feature patterns responsible for the high activations in each top image. The toolbox uses the visualization technique described in Section 2.3.5 to create the synthesized images, and the technique from Section 2.3.4 to produce the nine activation inducing feature patterns. Of all the visualizations, only the layer activations are computed in real time. The others are precomputed, due to the large cost involved in their creation or collection.

3.1.2 Visualization Tools as Influential Factors

When investigating the different visualization tools, we were most interested in which frameworks were supported and what functionality they offered. DIGITS have the widest support and can be used with networks built using Caffe, Torch and, come July, TensorFlow. In contrast, TensorBoard and the Deep Visualization Toolbox are restricted to a single framework each. To utilize TensorBoard, networks must use TensorFlow. For the Deep Visualization Toolbox, Caffe is the required framework. In terms of functionality, each of the three tools provide their own characteristic functionality, with a slight overlap of the most elementary visualizations.

A substantial difference of the Deep Visualization Toolbox compared to the other two tools, is that it can only be employed on fully trained networks. Rather than assisting the user in training, it focuses on deepening the understanding of the internal mechanisms of a network. TensorBoard and DIGITS, however, are both intended to be used during training, to help evaluate a network as it progresses. DIGITS is also capable of facilitating the whole creation and training process, and provides an organized system for managing datasets and models. Furthermore, DIGITS supports the use of fully trained networks.

Another distinction between the tools is the varying complexity of their visualization techniques. DIGITS provides the basic network visualizations, namely accuracy and loss plots throughout a training session. Additionally, it is able to visualize the layer weights and activations for a given input during network prediction. TensorBoard can also visualize the change in accuracy and loss, but focuses more on network metadata and statistics than the concrete values in the network. However, the most insightful visualizations are offered by the Deep Visualization Toolbox, which utilizes significantly more advanced techniques, such as deconvolutional networks and deep visualization.

When creating our own visualization tool, we build on the ideas of DIGITS, TensorBoard and the Deep Visualization Toolbox, aiming to bridge the gap between them. As we plan to visualize throughout the training process, we intend to adopt the basic visualizations of accuracy and loss found in TensorBoard and DIGITS. We also believe that the network management offered by DIGITS is highly valuable, and aspire to incorporate a similar feature in our tool. To be able to provide users with a deeper understanding of their networks, we intend to include the advanced visualization techniques from the Deep Visualization Toolbox as well. By employing the increasingly popular Keras API[9], we will be able to handle networks made with the Theano framework, which is not supported by any of the mentioned tools, and the TensorFlow framework, which is only supported by TensorBoard as of May 2017. In short, our tool will attempt to consolidate important functionality, and improve and extend framework support.

3.2 Case Study in Face Recognition

The proposed approach of exploiting facial expression in a face recognition network is a novel one. As a consequence, the earlier work related to the case study is scarce. Although we do not aspire to explicitly create a network that pushes the state of the art, we are interested in investigating a technique that might prove useful in future attempts to do so. This section therefore offers a brief overview of a selection of state of the art systems for facial recognition and their performance. Furthermore, we introduce an object classification network that has been repurposed for face recognition, and review its relevance to our case study.

The networks mentioned in this section are often described to be deep or even very deep. It is worth mentioning that the notion of depth in ANNs has changed somewhat as the research on such networks has progressed. With the existence of ResNet-152 [42], an accomplished 152-layer deep ANN, networks with less than 20 layers seem shallow in comparison. However, these smaller network are still commonly referred to as deep. For consistency, the practice is continued in this thesis.

3.2.1 Improving Face Recognition

The area of face recognition is widely researched, and the application of ANNs has enjoyed great success in recent years. A popular benchmark dataset is Labeled Faces in the Wild (LFW) [43], and in 2015 a network called FaceNet achieved 99.63% verification accuracy on the set [36]. While this performance is remarkable, even slight increases in accuracy are highly valued, as is evident when considering the previous state of the art, the DeepID3 network, which achieved 99.53% accuracy on LFW [38]. It is clear that the research area is in a state of diminishing returns, where considerable amounts of work is required to produce minor improvements. Further evidence of this can be found in earlier work, with the DeepFace network increasing state of the art accuracy from 96.33% to 97.35% [37]. Consequently, if the incorporation of facial expression data in a network produces even a slight rise in accuracy, the results are interesting. It is worth noting that not all of the promising results belong to ANNs, with the GaussianFace model obtaining 98.52% accuracy on LFW, being the first system to surpass the verification accuracy measured for humans on the dataset (97.53%) [44].

[9]https://keras.io/

3.2.2 Using ILSVRC Networks for Face Recognition

While all the aforementioned state of the art ANNs have their own distinct network architectures, common characteristics are that the networks are both convolutional and deep. This is also a distinction shared by several top performing networks entered in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [45] over the years [46, 47, 41]. These ILSVRC networks are interesting because of their powerful feature extracting capabilities and performance on varied object recognition in unconstrained environments. These qualities make them excellent choices for transfer learning networks. The use of deep CNNs in ILSVRC was initially inspired by the so-called AlexNet [40], which won the object classification category with a 15.3% test error rate against the 26.2% error rate of the closest competitor. The victory triggered an interest in what such networks were capable of, leading to successful applications in multiple computer vision fields, including face recognition.

In Parkhi et al. [48], several ANNs based on an ILSVRC network are tested in the face recognition domain. More specifically, the networks are modelled after the Visual Geometry Group (VGG) architecture, which was awarded second place in the object classification task of ILSVRC 2014 [47]. To train the face recognition networks, Parkhi et al. [48] constructed a dataset of celebrity images, consisting of 2622 identities with 1000 images each. The networks were tested on LFW and YouTube Faces Dataset [49], where the best accuracies achieved were 98.95% and 91.6%, respectively. These scores are comparable to those achieved by the state of the art networks DeepFace, DeepID3 and FaceNet, mentioned in Section 3.2.1.

3.2.3 Application to Case Study

For our case study network, we intend to employ the VGG architecture as a feature extractor. It has proven capabilities in both complex object classification and face recognition tasks, as well as having a fairly straightforward architecture. The latter is beneficial as lower complexity makes reasoning about experimental choices less complicated and reduces the possibility of unforeseen effects. However, VGG networks are prohibitively slow to train. To circumvent this issue, an option is to apply transfer learning to a fully trained version of a VGG network. The networks from Parkhi et al. [48] are prime candidates because of their impressive performance in a related problem area.

Chapter 4

Implementation

This chapter thoroughly describes the implementation process and the implementation itself, for both the visualization tool and the networks implemented for the case study. For clarity, all code examples in this chapter have omitted comments and code irrelevant to the example.

4.1 Visualization Tool

In this section we will present the implementation details of the visualization tool created as part of the thesis. We start with a summary of the related results from the Specialization Project, before explaining the system development methodology used in the implementation process. Then, we present the two quality attributes that have been focal points throughout the design and implementation phase, and state our choices regarding the technology used. Finally, we showcase the implementation details for the main parts of the visualization tool.

Note that this section does not explain how to install or use the visualization tool. This is described in Appendix A and Appendix B, respectively.

4.1.1 Prototype

This section presents the results from the Specialization Project in terms of the previously implemented prototype of the visualization tool, as well as some pointers on the planned improvements and extensions to be included for the thesis.

Description

The prototype provides all the basic user functionality for creating a user, and logging in and out, including validation of the user name and password. Once created, a user can upload Python scripts from their computer to the server, and tag them with various labels to make them easier to search for. The search bar is located on a page showing all uploaded scripts.

By selecting a specific script, the user can view its code, and add and delete tags associated with it. The user can also run the script, or delete it. A separate menu tab for the visualizations grants the user access to visualizations of the data produced during script execution. The visualizations available are the training progress and the layer activations, both presented on a single page. The training progress includes two separate plots showing the training accuracy and loss over the batches, while the layer activations are shown for each single layer of the script-defined network.

Regarding Planned Improvements and Extensions

The Specialization Project provided a list of major and minor extensions that would be useful to implement in an improved version of the visualization tool. Upon beginning the thesis work, the main focus of the thesis was shifted from the face recognition study to the visualization tool. Instead of the tool being created for the purpose of aiding our case study effort, the goal is now to release the code as open-source software for the benefit of other researchers. As a result, we have focused on properly implementing and testing the user functionality and visualization techniques for general networks. The tool supports fully sequential networks, but because of resource limitations, support for non-sequential models has not been added. There was also a shift in the use of the tool, namely that instead of a web application running on a server, it would be a desktop application that the user runs in a web browser. The user system is kept because it still has value in certain scenarios. For instance, if a lab provides a powerful computer used to train ANNs that several people have access to.

The move to a desktop application affected the relevance of certain planned extensions, and we have therefore decided to discard some of them. However, the two most important extensions were added: allowing a user to download a network, and implementing the saliency maps, deconvolutional network and deep visualization techniques. In addition, we have added more functionality, like stopping the execution of a script, using user uploaded images in visualizations, and viewing the console output produced by the script. The presentation of the visualizations have also been improved, making them more interactive and readable.

4.1.2 System Development Methodology

The development of the visualization tool has been carried out by a team of two, with no specific set of roles involved. On account of the small team size, a dedicated project manager was not deemed necessary. Both team members were involved in the design of the architecture and interface, as well as system implementation and testing. An efficient workflow was maintained by dividing the programming work into two main areas of responsibility:

1. Implementing visualization techniques and creating networks to produce example data.

2. Implementing visualization data processing and presenting the result to user interface.

The development process itself did not follow any strict guidelines, but included several elements from Agile methods[1]. It has been an iterative and incremental process, starting with the existing prototype of the system that only implemented two of the visualization techniques. Based on our own testing, as well as feedback from our supervisors, functionality was added and adjustments were made in order to obtain a new and improved prototype. This process was repeated until we reached a satisfactory system according to the initial requirements. An example of the iterative part of the process is the replacement of the old visualization library in favor of a new one that was better suited for visualization interactivity and streaming of real-time data. An example of the incremental part of the process is the shift to user-uploaded images for usage in certain visualization techniques, which was added after supervisor feedback. Other concepts from Agile development that were applied are code review, pair programming, and the use of a backlog to get an overview of the requirements.

4.1.3 Focus Quality Attributes

In addition to functional requirements, we wanted to incorporate some non-functional requirements as well. To that end, we have selected two quality attributes that we consider to be of high importance. Quality attributes are overall factors that represents areas of concern which impact the application as a whole. In our case, we have determined that the most important factors are modifiability and usability.

Modifiability

An important aspect of the visualization tool's system design was to ensure that certain parts could be easily replaced. For instance, a user should be able to alter the system to employ a different deep learning library than Keras without much difficulty. This requires thorough consideration when designing the architecture, and typically calls for a module-based architecture with loosely coupled modules, meaning that each module should interact with as few of the other modules as possible.

Usability

Not only should the interface be simple and easy to navigate for the user, but the actual installation and setup of the visualization tool should be straightforward. In addition, the tool's requirements for the user scripts should be easy to fulfill. A comprehensive user manual and documentation of the API is the key to obtaining this kind of usability. Preferably, we would want any ANN to run in our program without issues, but the extensive variety of network architectures makes it incredibly difficult to acquire such generalization. Consequently, we prioritized supporting the most common architectures.

[1]https://www.agilealliance.org/

4.1.4 Technology Decisions

This section presents the technology used in our implementation, as well as justification of why they were selected and identification of any potential drawbacks.

Flask

Flask[2] is a web framework for Python, used to build simple web applications. It is a micro-framework, meaning that it has a minimal amount of dependencies on external libraries. It provides only the basic tools needed to create a web application, and is therefore very lightweight. However, there exist many plugins that can be added for an increased range of functionality. Since the focus of our application is not the web page itself, but rather its capabilities in terms of visualization techniques, Flask was the perfect choice. Together with a selection of plugins, it allowed us to quickly set up an application and implement a basic user system and upload functionality.

SQLite

SQLite[3] is a simple and lightweight relational SQL database engine. In contrast to many other databases, it is embedded into the application. It is often used to provide local data storage for desktop applications. An embedded database improves application performance, reduces cost and complexity, and improves reliability. The visualization tool does not require any advanced database functionality, nor does it store large amounts of data or utilize many database connections simultaneously. The lightweight SQLite therefore represents a suitable choice. Additionally, it is uncomplicated to set up, and could be easily replaced with a more advanced database system at a later time, if needed.

Keras

Keras[4] is a high-level neural networks library that can run on top of either TensorFlow[5] or Theano[6]. Its purpose is to facilitate quick and easy creation of networks and, by extension, rapid experimentation. Keras offers a wide range of network functionality, enveloped in a straightforward and well documented API. It also has a strong modifiability basis, making it easy for users to create functional extensions using custom modules. The library contains several callbacks, which are sets of functions to be applied at given stages of the ANN training procedure. It also allows for creating custom callbacks, which is advantageous when we want to apply various visualization techniques at certain stages of training. Consequently, we chose to develop our visualization tool with support for Keras including both the TensorFlow and Theano backend. The use of Keras is strictly limited to the callbacks, to ease a potential substitution of the deep learning library.

[2]http://flask.pocoo.org/
[3]https://www.sqlite.org/
[4]https://keras.io/
[5]https://www.tensorflow.org/
[6]http://deeplearning.net/software/theano/

Bokeh

Bokeh[7] is a Python-based interactive visualization library for modern web browsers. It can be embedded into a Flask application without difficulty, as described later in the chapter. The library is easy to use, and provides the flexibility of adding interactions and advanced customization without much effort. Another benefit is that it allows for streaming large data sets and plot them live, a highly useful feature for our application. However, Bokeh is still in an early state of development, which makes it more susceptible to bugs, and will in some cases limit its functionality.

4.1.5 Overview of the Architecture

As seen in Figure 4.1, the system can be divided into five separate modules: the user storage, Bokeh, Flask, Keras and the database. The Flask module represents the core of the application. It defines the routes (URLs), models and forms of the web pages, tying the functionality together with HTML templates, CSS and JavaScript into a simple web interface. The user storage contains all the scripts and images that are uploaded through the Flask application, as well as the visualization data produced during network training. The database stores the metadata concerning the uploaded scripts, such as file owner, upload timestamp, and the file path. The Keras module includes the custom callbacks created for the visualization techniques. These write data to the files saved in the user storage. To include a specific visualization technique, a user simply includes the corresponding callback in his or her script. It is the Flask application that implements the functionality for executing these scripts. Finally, the Bokeh module uses the data located in the user storage to create interactive visualizations, and the Flask application handles how these are displayed in the web interface.

A package diagram in Figure 4.2 shows a more low-level view of the architecture. There are three packages: `custom_keras`, `custom_bokeh` and `visualizer`, which correspond to the Keras, Bokeh and Flask modules, respectively. In this case, a package corresponds to a folder inside of the main project folder. The folders and the files they contain are displayed, as well as the dependencies between them. The packages will be explained in further detail in the following sections.

4.1.6 The Flask Application

Figure 4.3 shows an overview of the structure of the files that are included in the Flask application. The `static` folder serves all the CSS and JavaScript files, while the `templates` folder contains Jinja2 [8] templates that Flask can render to generate the HTML. The `__init__.py` file contains the instantiation of the Flask application, as well as partial setup for the database and login system. The `config.py` file configures certain application parameters, such as setting the system commands necessary for running Python scripts. The database, `visualizer.db`, and the `user_storage` folder are also seen in the file structure. These will be further explained in Section 4.1.7 and Section 4.1.8,

[7]http://bokeh.pydata.org
[8]http://jinja.pocoo.org/

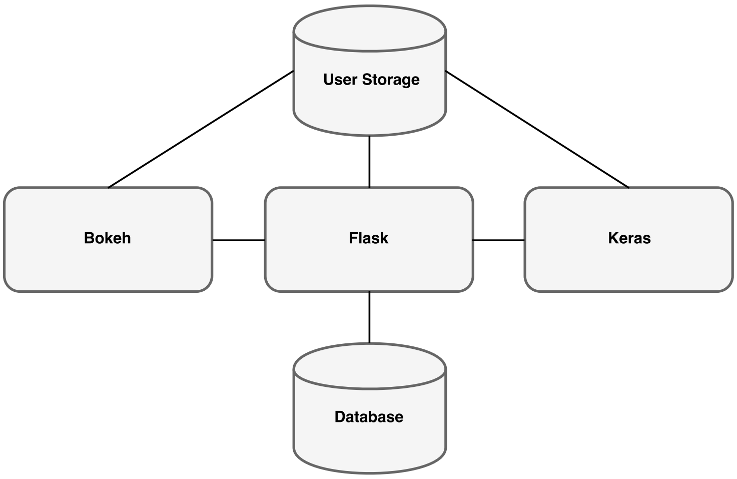


Figure 4.1: Overall Architecture of the System

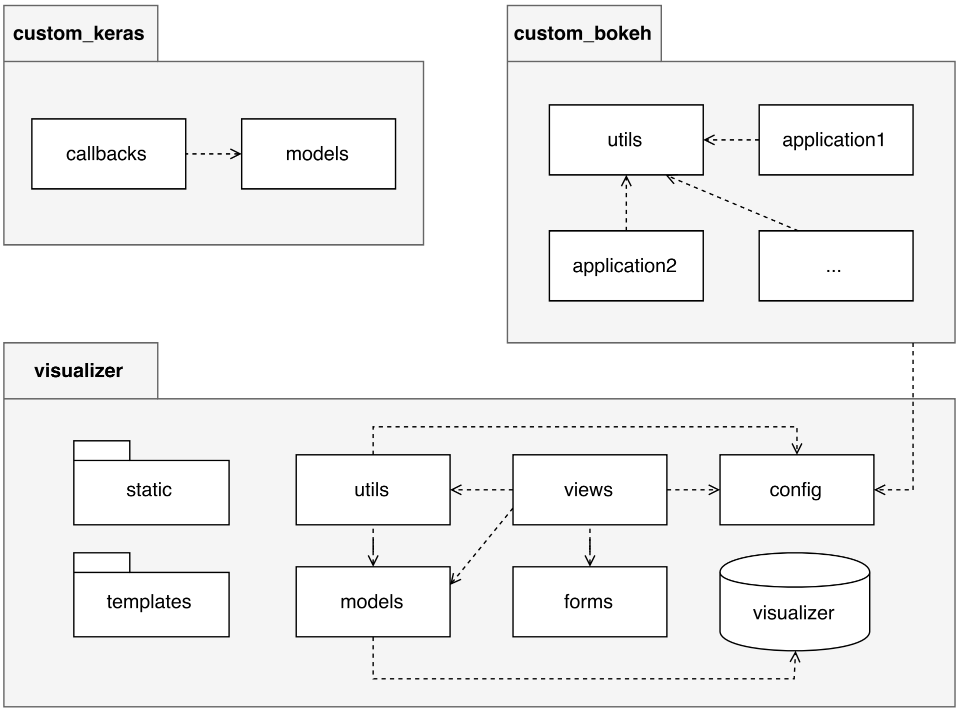


Figure 4.2: Package Diagram

respectively.

```
/visualizer
   /static
      /css
      /fonts
      /js
   /templates
      /create_user.html
      /home.html
      /layout.html
      /...
   /user_storage
      /...
   /__init__.py
   /config.py
   /forms.py
   /utils.py
   /models.py
   /requirements.txt
   /views.py
   /visualizer.db
```

Figure 4.3: Flask project file structure

There are three main files implementing the functionality of the Flask application: `forms-.py`, `models.py` and `views.py`. They define all the forms, models and views, respectively, of the web interface. A form is used to collect user input, for instance when registering a new user, or uploading a script. Models are used to create database objects, which are explained in the next section. Lastly, a view defines a URL endpoint, or a route, as well as what HTTP requests it answers to, and determines what the system should do when a user navigates to that URL. There also exists a `utils.py` file that contains various utility methods.

An example of a view from `views.py` is shown in Code 4.2. Line 1 defines the route of the view and the HTTP requests to answer to, which is GET and POST. GET is used for returning the user register page, while POST is used when the register button is actually clicked. The code illustrates the use of a form in line 3, in this case for creating a new user, which will contain the chosen username and password of the new user. Line 5 checks if all the form fields are valid when a user has submitted a POST request, while the next line alerts the user and halts the process if the username is already taken. If not, line 9 creates a User database entry consisting of the username and password and adds it to the database. Note that the password is salted and hashed upon entry creation, for security reasons. Line 13 redirects the user to another view, namely the login view. If the user has not submitted a form, or the submission was unsuccessful, line 15 ensures that the correct template for the user registration page is rendered.

```
@app.route('/create_user', methods=['GET', 'POST'])
def create_user():
  form = CreateUserForm()

  if form.validate_on_submit():
    if not unique_username(form.username.data):
      flash('Username is already taken', 'danger')
    else:
      db.session.add(User(form.username.data, form.password.data))
      db.session.commit()

      flash('User successfully created', 'success')
      return redirect(url_for('login'))

  return render_template('create_user.html', form=form)
```

Code 4.2: View for creating a user

4.1.7 Database

As mentioned earlier, the database is quite simple, and does not store a large amount of information. Its purpose is to handle the user system and store useful meta data about the uploaded scripts. It consists of three models, with attributes and connections as seen in Figure 4.4. The `User` model contains necessary information of a user. The `FileMeta` model contains the meta data of the scripts, as well as a connection to its owner. There is also a separate `Tag` model, that can be associated to zero, one, or several scripts.

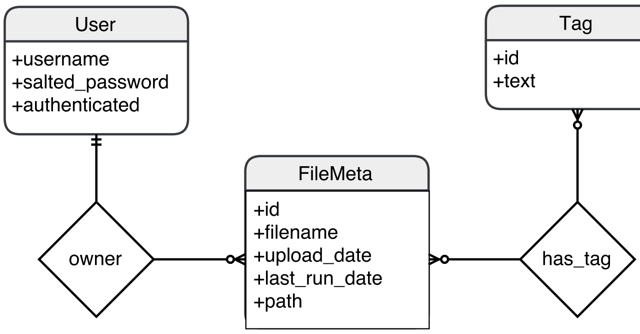


Figure 4.4: ER-diagram of the database

4.1.8 User Storage

The user storage is where the scripts are uploaded to, and where the visualization data is stored. The structure of the user storage can be seen in Figure 4.5. The top level holds the different users of the system. The second level contains the various scripts that a user has uploaded. At the next level, the script file itself is stored, as well as a text file containing the console output of the script. There is a folder called `images` that contains the image that the user has uploaded to be used in the visualizations. The `networks` folder contains the most currently saved network created by the script. The folder called `results` stores all the data produced by the various callbacks. The files are all given descriptive names so that it is self-evident which file corresponds to which visualization technique. The format of the visualization files are presented in Appendix D.

```
/user-storage
  /<user1>
    /<script1>
      /images
        /<image>
      /networks
        /<network>
      /results
        /deconvolutional_network.pickle
        /deep_visualization.pickle
        /layer_activations.pickle
        /saliency_maps.pickle
        /training_progress.txt
        /training_progress_val.txt
      output.txt
      <script1>
    /<script2>
      /...
  /<user2>
    /...
```

Figure 4.5: User storage structure

4.1.9 Running a Python Script

The visualization tool allows a user to both start and stop the uploaded Python scripts. Functionality for running a script is implemented as a utility function, and is done by executing `python [file_path]` in a new subprocess, as seen in Code 4.3. The actual Python command is determined by a configuration variable, since it is dependant on the user's Python installations (some systems use `python3` for Python 3.x and simply `python` for Python 2.x). The `stdout` argument specifies the standard output file handle, which is in this case a text file that will be displayed in the web interface, so that the user is given easy access to the console output of the script. Note that even though Figure 4.1 shows a connection between Keras and Flask, the real connection is really between Python and Flask. A user could easily upload a Python script that uses the scikit-learn library[9] or pure TensorFlow, and running such a script would not cause any issues. However, if someone wanted to adapt the visualization tool to a whole different programming language, this section of the code would need to be replaced.

```
with open(get_output_file(get_current_user(), basename(file_path)), 'w') as f:
  p = subprocess.Popen([PYTHON, file_path], stdout=f)
```

Code 4.3: Running a Python script using subprocess

4.1.10 Custom Keras Callbacks

The custom callbacks are the only part of the visualization tool that are specific to Keras. As mentioned, a callback is a set of functions to be applied at given stages of the ANN training procedure. Keras provides several predefined callbacks, but also includes the possibility of creating custom variations. We have taken advantage of this functionality, and have created a callback for each visualization technique, as well as for other useful functionality. An overview of the custom callbacks, and how to adapt them to your script, is thoroughly explained in Appendix C. In addition, we will also showcase certain parts of their implementation in this section. The implementation of the visualization functionality in the callbacks follows the processes explained in Section 2.3 quite strictly. Consequently, these parts will not be examined any further here. We will, however, explain the implementation details of the deconvolutional network that are not covered by the background theory.

CustomCallbacks - A Wrapper Class for the Custom Callbacks

Every custom callback requires the system path to the folder containing the script they are to be used in. In addition, a subset of the callbacks share similar, optional arguments. To simplify the user's process of adding callbacks to their code, we have created a wrapper class that takes all the common arguments on its instantiation. The wrapper class also provides methods for registering each callback with their associated arguments, excluding the shared ones, and a method for getting the list of registered callbacks. A user can apply their selected callbacks by passing this list to the `callbacks` argument of the `.fit()` method of a Keras model.

An excerpt of the implementation of the wrapper class is shown in Code 4.4. There you can see the initialization of an instance of the class, which instantiates the necessary and optional variables in lines 3-10. The argument named `file_folder` is the path to the folder of the script the callbacks are applied to, denoted as `<script1>` and `<script2>` in Figure 4.5. This is needed to know where the callbacks should store their produced data. The arguments `custom_preprocess` and `custom_postprocess` are functions that can be defined by the user to pre- and post-process images to be used in visualization.

[9] http://scikit-learn.org/stable/

The argument named `base_interval` is the batch interval that each callback is applied at, unless overridden at callback registering. The callbacks vary in their computational complexity, and some of them take significantly longer to execute than others. These have their own interval argument that supersedes the base interval.

The excerpt also shows two of the register methods of the wrapper. The first one, named `register_training_progress` , seen in line 15, simply creates a new instance of the `TrainingProgress` callback, using the previously defined file folder as a parameter, and adds it to the callback list. The second method, named `register_saliency_maps`, in line 18, does the exact same thing, except that it also uses the aforementioned interval argument. If the user does not specify an alternate interval upon registering, the interval of the callback is set to the base interval. The pre- and postprocess functions, if specified, are also passed to the callback. The method in line 12 returns the list of all added callbacks.

```
 1  class CustomCallbacks:
 2
 3    def __init__(self, file_folder, custom_preprocess=None,
 4                 custom_postprocess=None, base_interval=10):
 5
 6      self.file_folder = file_folder
 7      self.custom_preprocess = custom_preprocess
 8      self.custom_postprocess = custom_postprocess
 9      self.base_interval = base_interval
10      self.callback_list = []
11
12    def get_list(self):
13      return self.callback_list
14
15    def register_training_progress(self):
16      self.callback_list.append(TrainingProgress(self.file_folder))
17
18    def register_saliency_maps(self, interval=None):
19      if interval is None:
20        interval = self.base_interval
21      self.callback_list.append(SaliencyMaps(self.file_folder,
22                                             self.custom_preprocess,
23                                             self.custom_postprocess,
24                                             interval))
```

Code 4.4: Wrapper class for custom callbacks

Example Implementation of a Callback

The implementation of the `LayerActivations` callback can be seen in Code 4.5. Specific functionality is omitted, as the purpose is to showcase the general approach. As all the custom callbacks, it is an extension of the base abstract class `keras.callbacks.Callback` as seen in line 1. The initialization takes all necessary and optional arguments, and set the attributes accordingly. The folder to save the data is obtained by adding 'results' to the `file_folder` argument. The counter variable is set to one less than the interval so that an initial visualization will be produced. Line 13 starts the process of getting the visualization image and converting it to the correct format. In line 19, the preprocess function is applied if the user has specified one.

The `keras.callbacks.Callback` abstract base class defines various methods that are called at specified events, such as `on_batch_end` and `on_train_begin`, that are used in this example. In other custom callbacks, `on_epoch_end` is also used. In line 24, at the beginning of the training process, we call `on_batch_end` with the batch number 0 to produce the initial visualization. The counter value is increased in line 28, and line 31 checks whether it is time to perform the computation of the visualization data. The specific functionality varies from the different visualization techniques. Finally, line 35 creates the correct file for storing the data, in this case in a pickle file, and the counter is reset in line 39. Note that some of the callbacks also make use of a postprocess function, which would be applied to the visualization output before saving it.

Deconvolutional Network Implementation Details

Using the deconvolutional network visualization technique requires a deconvolutional network which can reverse the mapping from input to features that the convolutional part of the original network performs. In the implemented callback for this technique, the deconvolutional network can be automatically generated for simple models. The generation process is restricted to building a model corresponding to a continuously convolutional part that represents the first component of the original network. For example, if a fully connected layer separates two convolutional parts, only the first part is considered when constructing the deconvolutional network. The generation is also limited to convolutional parts which contain layers using convolution or max pooling. Before the process can start, the convolutional part is identified by examining the original network. This part is explored from top to bottom, while the deconvolutional network is being built from the bottom up. For every convolutional layer in the original, the appropriate deconvolutional substitute is added. Concurrently, a map concerning corresponding original and deconvolutional layers is being created. If a deconvolutional model cannot be created, the user must build the model and pass it to the callback.

To approximately reverse the effects of max pooling layers, the auto-generated deconvolutional networks use a custom Keras layer we have created called MaxUnpooling2D. These layers are used instead of the switch variables described in Section 2.3.4, as this would require interfering with the uploaded networks, which could have unforeseen consequences. MaxUnpooling2D layers are image specific, which in turn make the deconvolutional mod-

```python
class LayerActivations(Callback):

  def __init__(self, file_folder, exclude_layers=EXCLUDE_LAYERS,
               custom_preprocess=None, interval=10):

    super(LayerActivations, self).__init__()
    self.results_folder = join(file_folder, 'results')
    self.interval = interval
    self.counter = interval - 1

    self.exclude_layers = exclude_layers

    images_folder = join(file_folder, 'images')
    img_name = listdir(images_folder)[-1]
    img = Image.open(join(images_folder, img_name))
    self.img_array = img_to_array(img)

    if custom_preprocess is not None:
      self.img_array = custom_preprocess(self.img_array)

    self.img_array = np.expand_dims(self.img_array, 0)

  def on_train_begin(self, logs=None):
    self.on_batch_end(0)

  def on_batch_end(self, batch, logs={}):

    self.counter += 1
    layer_tuples = []

    if self.counter == self.interval:

      # specific functionality, removed to simplify example

      with open(join(self.results_folder,
                     'layer_activations.pickle'), 'wb') as f:
        pickle.dump(layer_tuples, f)

      self.counter = 0
```

Code 4.5: Example implementation of a callback els using them image specific. They compute the pooling regions used in the original max pooling based on the region size, strides, padding, as well as the shape of the input and output. These regions are computed differently for Theano and TensorFlow, depending on the aforementioned values. Then, for each region, the layer identifies which position the original max value was selected from by examining the original input and output. Subsequently, a zero matrix the size of the original input is created, and the max positions are filled with their corresponding values in the MaxUnpooling2D input. Thus, the functionality of the switch variables have been reproduced, without altering the original network.

When visualizing via the web interface, the deconvolutional network will either identify and visualize the feature maps that maximally activate on the user uploaded image, or visualize the feature maps directly specified by the user. Which of these options are chosen are based on the arguments passed when instantiating the callback. The arguments are explained further in Appendix C. The callback only allows visualization of feature maps from a single layer, but there is no limit on the amount of feature maps from the chosen layer. In addition to these alternatives, we have also implemented functionality that produces visualizations for a set of images that maximally activate certain feature maps. The feature maps can either be specified directly or a given amount can be randomly sampled. A large set of images will then be examined for each of these feature maps, computing the activation for each one. A predefined amount of images are selected for each chosen feature map, and their deconvolutional visualizations are produced. The max activation images represent samples that specifically contain the feature that the map's corresponding filter is looking for, and the visualizations of these images will highlight that feature. The image set from which the max activation images are sampled is based on the 2011 ILSVRC object classification image set [45]. This functionality is not available in the tool, however, as it was found to be prohibitively slow in its current state. Although it was implemented with a user specified limit on how many images to process, it should be heavily optimized before being integrated into the web interface.

Unfortunately, the deconvolutional network visualization technique is not available for Theano at the time of writing. This is due to an error within the Keras API regarding the use of transposed convolutional layers with Theano as a backend.

4.1.11 The Bokeh Server

The visualizations are displayed in the user interface using the Bokeh visualization library. The most flexible approach of embedding these into the Flask application is to make use of the Bokeh Server. A Bokeh application is created for each visualization technique, and these are served by the server using the `bokeh serve [applications]` command. The `applications` argument is a list of paths to the various applications, separated by spaces. The server uses the application code to create sessions and documents. A document is Bokeh's organizing data structure, containing all the models and data needed to render the application in the browser.

The Bokeh Server is embedded into the code of the Flask application as seen in Code 4.6. This method is called whenever a user selects a tab that contains a visualization. The base URL of the Bokeh server is `BOKEH_SERVER`, imported from the config file, and `app_path` is the corresponding relative path to the selected visualization, for instance 'training_progress'. The full URL of the training progress Bokeh application is thus `BOKEH_SERVER` joined together with `app_path`. The server is loaded in line 3, returning a script tag that will replace itself with a Bokeh plot when placed in an HTML template. The Bokeh application will need to know the filename and the username in order to locate the correct data file to visualize. Bokeh allows for passing HTTP request arguments to its applications, but has not yet implemented a way of passing these in `autoload_server`[10]. Therefore, we need to manually insert the filename and username into the script tag, as seen in line 5 through 11. Finally, the script can be returned and is further passed on to an HTML template. The complete process, starting from a user clicking on one of the visualization tabs, to the user actually viewing the visualization, is illustrated in the sequence diagram in Figure 4.6.

```
1  def get_bokeh_plot(filename, app_path):
2
3    script = autoload_server(model=None, url=urljoin(BOKEH_SERVER, app_path))
4
5    params = {'user': get_current_user(), 'file': get_wo_ext(filename)}
6
7    script_list = script.split('\n')
8    script_list[2] = script_list[2][:-1]
9    script_list[2] += '&' + urlencode(params) + '"'
10
11   return '\n'.join(script_list)
```

Code 4.6: Embedding the Bokeh server in Flask

4.1.12 Bokeh Applications

As mentioned, a Bokeh application is created for each visualization technique, resulting in the following five applications:

- `deconvolutional_network.py`
- `deep_visualization.py`
- `layer_activations.py`
- `saliency_maps.py`
- `training_progress.py`

[10]This functionality will be available in the next version of Bokeh, 0.12.7, in response to our feature request at https://github.com/bokeh/bokeh/issues/5992

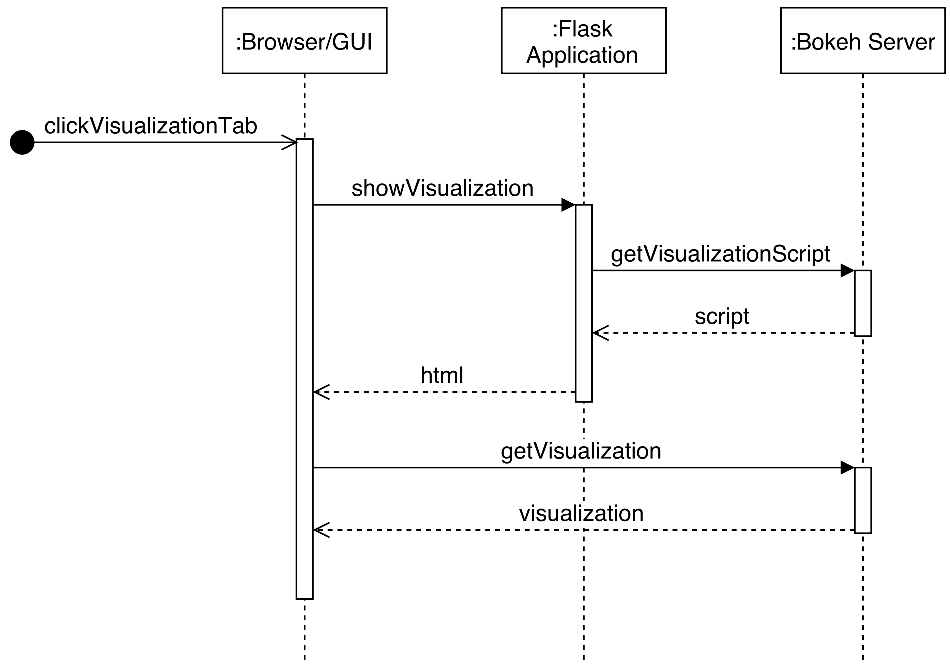


Figure 4.6: Sequence diagram of getting a visualization from Bokeh Server

We will not describe the implementation of the applications in detail. Most parts of the code deals with the creation and updating of figures using functions provided by Bokeh, which is not particularly interesting for the thesis. Rather, we will present some of the most important concepts of the implementations. The resulting visualization can be seen in Chapter 5.

Handling the Visualization Data

All of the Bokeh applications begin with the same lines of code, shown in Code 4.7. The document variable in line 1 holds the current Bokeh document that we are working on. In lines 2-5, we get the HTTP request arguments from the Flask request. These arguments are used to find the path in the user storage corresponding to the file we are going to visualize data for, as seen in line 7. `UPLOAD_FOLDER` is the path to the user storage folder, imported from the configuration file. Some of the Bokeh applications also display the original image used in the visualization. This image is found using a similar method as for the visualization files.

The data loaded from the visualization file is processed to a format that is beneficial for presentation. The processing depends on the specific visualization technique, and the format of the visualization files can be seen in Appendix D. For instance, data for the training progress is transformed from rows of x- and y-values into separate sequences of x- and y-values. The other techniques mostly output image data, which needs to be converted into

```python
document = curdoc()
args = document.session_context.request.arguments

file = args['file'][0].decode('ascii')
user = args['user'][0].decode('ascii')

results_path = join(UPLOAD_FOLDER, user, file, 'results')
```

Code 4.7: Getting the data from the visualization files

2D arrays, since Bokeh does not handle 3D arrays for images. For layer activations, the results include a large amount of images for the convolutional layers. These are stitched together into a grid of images for each layer. When the data has been properly processed, it is stored in a `ColumnDataSource`, a Bokeh object that stores data to be used in a figure. In our case, its main purpose is to allow for simple streaming of data. When associating a ColumnDataSource with a figure, updating the data of the `ColumnDataSource` automatically triggers an update of the figure. This approach is much more efficient than creating a new figure every time the data is updated. The update is performed by adding a periodic Bokeh callback to a function that reads the visualization file and updates the data of the `ColumnDataSource`. The interval of this update is defined by the Flask configuration file.

4.2 Case Study in Face Recognition

In this section, we describe the implementation details concerning the case study. To begin with, we explain how expression information can be useful, and then present ideas about how experimental networks can utilize this information. Subsequently, we provide insight into our methods for obtaining satisfactory networks based on these ideas. We then examine a pretrained model, which was used as a feature extractor for the case study networks. Next we introduce the dataset used to investigate our theory, and how the images of this set were preprocessed. Finally, nine different architectures are suggested, three for each network variation. These architectures are considered to be separate from the pretrained network, but using its feature output as input. Note that these architectures are all specified using the Keras 2 API terminology.

4.2.1 Exploiting Expression Information

The way facial expressions alter the facial geometry is predictable, both in a specific and general sense. If a specific person conveys the same facial expression several times, the expression will consistently affect a certain set of features and always in a similar manner. For instance, if an image of a happy person includes feature variations such as a widened mouth, narrower eyes, and dimples, another image of that person being happy will likely include the same variations. Some alterations can also be expected to appear generally.

While dimples are not a universal trait, a smile, causing a wider mouth, can be anticipated to be commonplace in facial expressions conveying happiness. Knowing or recognizing the facial expression of a person would then reveal some information about how the facial features have transformed in relation to the person's neutral features. This case study examines how to exploit this predictability by including facial expression data in the ANN training procedure. If a face recognition system could learn how these expressions affect facial geometry, then the system could be able to match the dissimilarities in the facial features with the changes it knows the facial expression produces. This way, we can move away from trying to overcome the expression challenge through invariance, and instead utilize the data about facial expressions to increase network performance.

4.2.2 Experimental Network Idea

For the case study, we wanted to examine if the performance of a face recognition network could be improved by incorporating facial expression data. To examine this idea, we considered three different neural network approaches for facial recognition. The first network was used as a standard to be measured up against, with no utilization of expression data. It simply receives an input image and emits an identity classification result as output, determining who the individual in the image is. The second network receives the facial expression of the person in the image as input, in addition to the input image. The output will be similar to the standard network. This network has the potential to use the expression data explicitly, to learn how the expressions alter which facial features match the various identities. The third network outputs both the identification classification result and a facial expression classification for the input image person, without receiving any extra input. By trying to classify both identification and expression simultaneously, the network has the potential to learn how the facial features and expressions relate to each other. The network would then implicitly learn how the expression input alters the facial features, which could improve identification classification. The performance of each network is measured by its total classification loss and accuracy. To simplify the training of these networks, a pretrained network was used as a feature extractor. A large, shared part of the networks is then already fully trained, leaving only the classification part, greatly decreasing the time required for training. Reducing the training time allows for more extensive experimentation.

4.2.3 Method

When creating the case study networks, we started by identifying advantageous baseline structures. To examine the performance, we used the tool described in this thesis to visualize the training progress, monitoring loss and accuracy for the training and validation sets. For the architectures with an extra output, these metrics were read manually from a log file, as the tool does not support training progress visualization for networks with multiple outputs. The other visualization techniques were not utilized due to the incompatibility with non-sequential networks, the homogeneity of the dataset, and the classification importance of relatively fine details.

The architectures that were investigated varied in depth and width, i.e. in the amount of units per layer. In addition, we explored different levels of regularization strength, which was determined both by how many layers experienced dropout and how high the rate of dropout was for each layer. We also looked into how different training hyperparameters affected the network, observing how they learned using various epochs and batch sizes, as well as different adaptive learning algorithms and initial learning rates.

After establishing a selection of satisfactory baseline architectures, we used them as starting points for the creation of the experimental networks. If any of the adaptions showed promise, we explored them further by adjusting depth and width, and experimenting with training length and the initial learning rate. For every new architecture, we kept any that offered improvements, quickly discarding those that failed to enhance performance, because of the large amount of potential configurations. Specific to the architectures utilizing an extra input, we investigated how the networks reacted to injecting the input at various locations. For the architectures that were required to output an extra classification result, we varied the amount of layers that were connected to both output layers and the amount of layers that were only connected to one of them. If we discovered an experimental network that performed admirably, but lacked a baseline counterpart, we also tested the corresponding baseline architecture. Through these incremental improvements, we discovered the final architectures described in Section 4.2.6.

4.2.4 Pretrained Model

As described in Section 2.4, transfer learning is an appropriate method for training an ANN without extensive training time or large amounts of training data. To employ transfer learning, a suitable pretrained model is needed. Since our case study deals with face recognition, it would be beneficial to use a CNN that was previously trained for such a purpose. The network would then already be able to detect specific facial features, like mouths and eyes. A suitable model is the 16 weight layer D configuration of the VGG-Face CNN [48]. The weight values of this configuration, as computed in Parkhi et al. [48], has been made available[11] and converted to use with the Keras framework[12]. This particular configuration will be referred to as the VGG-Face network from here on out. The original implementation used Caffe[13], but we will be using a Keras implementation converted from the original Caffe network. The Keras version allows us to utilize the visualization tool for managing the various networks created and visualizing certain data. We describe the architecture here, with the full implementation found at GitHub[14].

An overview of the pretrained network architecture is shown in Table 4.1. Here, the network has 25 layers, including the input layer. However, only 15 of the layers have weights associated with them. If we add the excluded classification output layer, the number rises to 16. This architecture is based on the network that is popularly called VGG-16 [47],

[11]http://www.robots.ox.ac.uk/ vgg/software/vgg_face/
[12]https://github.com/rcmalli/keras-vggface
[13]http://caffe.berkeleyvision.org/
[14]https://github.com/rcmalli/keras-vggface

which was originally applied to ILSVRC [45]. The architecture as demonstrated praiseworthy performance in face recognition and ILSVRC object classification, which both require excellent feature extraction capabilities. The feature extraction proficiency is made possible by the stacked convolutional layers, with max pooling layers to obtain some spatial invariance. All the convolutional filters are of size 3x3, which for the first layer is enough to capture small, simple features. The complexity of the features grows with the depth of the layers, each layer capturing features made up of lower level features. Depth also increases the effective receptive fields of the layer units, as they become indirectly connected to a larger number of input units. The fully connected layers are indirectly connected to every input unit, and are used to combine the extracted features of the convolutional layers to create even more complex and invariant feature representations.

4.2.5 Dataset and Preprocessing

Based on the restrictions regarding the dataset described in Section 1.5.2, as well as the review of potential datasets conducted in the Specialization Project, the Indian Movie Face Database (IMFDB) [50] was selected to be used in the case study. The database consists of 34,512 images of 100 actors collected from Indian movies, captured in an unconstrained setting. All images have been selected and labeled manually, and the labels include both identification and expression. The available expressions are the six basic expressions described in Section 2.5, in addition to a neutral expression, giving a total of seven possible values for the expression label. The dataset is structured so that it contains a folder for each of the actors. These folders again contain a folder for every movie where actor images were extracted. A movie folder contain an image folder and a text file with all image names and their corresponding labels.

After downloading the dataset, we received some errors while trying to process the text files. Thus, we created a script that traversed the text files of the dataset and alerted us of any peculiarities. Some of the entries were missing their expression labeling and these were removed entirely. Others were just improperly formatted, which was easily fixed. This process reduced the number of images, resulting in a dataset with 30,091 images of 98 actors. The dataset was then split into three sets consisting of 70%, 20% and 10% of the images. These sets were used for training, validation and testing, respectively. The images were already cropped with a tight bounding box around the faces, thus the only preprocesssing needed to use the images with the VGG-Face network were scaling and padding. The images were padded to squares with black, and scaled to an appropriate size of 130x130 pixels. The size was selected on the background of the mean image width and height of the dataset, which was calculated to be 102 and 129, respectively.

Although IMFDB provides everything we need in terms of labels, the quality of the dataset is debatable. When processing the dataset, we discovered that the smallest image is only 8x11 pixels in size. In comparison, the largest image is 852x893 pixels. Some of the images are small and of poor quality, but since having a large amount of images was a crucial factor to our case, we decided to keep all of them hoping that the general overall quality would be good enough. Another issue is the reliability of the expression labeling. As the

Layer #	Name	Layer Type	Units
0	input	InputLayer	224x224x3
1	conv1_1	Conv2D	224x224x64
2	conv1_2		
3	pool1	MaxPooling2D	112x112x64
4	conv2_1	Conv2D	112x112x128
5	conv2_2		
6	pool2	MaxPooling2D	56x56x128
7	conv3_1	Conv2D	56x56x256
8	conv3_2		
10	conv3_3		
11	pool3	MaxPooling2D	23x23x256
12	conv4_1	Conv2D	23x23x512
13	conv4_2		
14	conv4_3		
15	pool4	MaxPooling2D	11x11x512
16	conv4_1	Conv2D	11x11x512
17	conv4_2		
18	conv4_3		
19	pool5	MaxPooling2D	5x5x512
20	flatten	Flatten	12800
21	fc6	Dense	4096
22	fc6/relu	Activation	
23	fc7	Dense	
24	fc7/relu	Activation	

Table 4.1: Pretrained network architecture. The last two layers of the original model have been excluded, as these form the classification output and are therefore irrelevant. Note that the layer type refers to the Keras 2 API. All Conv2D layers use 3x3 filters, with single strides, same convolution padding, and ReLU activation functions. All MaxPooling2D layers use 2x2 pooling regions, with double strides and no padding. Flatten layer simply flattens input. Both Dense (fully connected) layers have linear activation functions, but are followed by an Activation layer using ReLU activation functions.

images are captured in an unconstrained setting, it may be problematic to assign appropriate expressions to every image. There are many ways to express an emotion, and the classification of an emotion based on a person's facial expression is a subjective process. A random selection of images from the dataset, labeled with each of the seven expression values, can be seen in Figure 4.7. It is evident that some of the labels seem more obvious and fitting than others. However, datasets captured in constrained settings were found to contain a significantly smaller number of images, as they all need to be captured manually. Unconstrained images are more easily retrieved in large amounts exploiting existing images on the Internet or still images from movies. Another advantage with employing an unconstrained dataset like IMFDB is that the pretrained model was trained on such

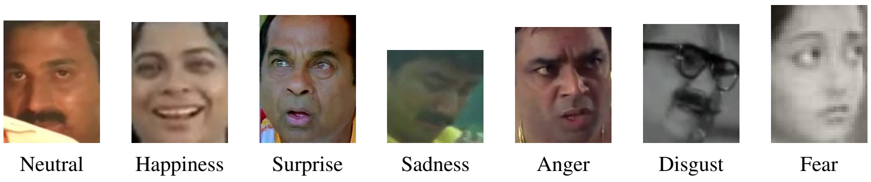


Figure 4.7: A labeled selection of images from IMFDB, one for each expression.

images, and it is convenient to utilize a similar dataset when conducting transfer learning.

4.2.6 Case Study Networks

All of the case study networks use the pretrained network as a fixed base. Hence, they can be viewed as separate networks whose input is shaped similarly to the pretrained network's output. We therefore describe the different architectures for the case study networks with no explicit connection to the pretrained network and consider the input to be a feature vector of size 4096. Should any of the architectures defined here be applied to images, they would need to be combined with the pretrained model. For each of the three approaches, we have included three architectures. For every approach, the three architectures consist of a minimal architecture, the architecture with the best performance, and an interesting alternative to the preferred architecture. The minimal and alternate architectures are included as comparisons to the preferred one, to help determine what makes the preferred architecture excel.

It is worth noticing that every selected architecture has a dropout layer right after the input layer. This was added as a standard because of the most basic baseline, where only an output layer was added, which quickly achieved 100% training accuracy and a very low loss value. The validation accuracy and loss, however, were stuck at less beneficial values. Naturally, overfitting was suspected, and while it became apparent in the later stages of training, the main issue was the lack of progress. By utilizing dropout on the feature input, the network had a harder time fitting perfectly to the data, and it showed a slight improvement in the validation metrics. Generally, the architectures are heavily regularized with dropout due to overfitting concerns, as it had a tendency to arise in architectures with less utilization of the dropout technique. The tendency to overfit is most likely caused by the relatively small size of the dataset used. In turn, this tendency restricted architecture exploration, as deeper architectures systematically overfitted and had unsatisfactory performance.

Baseline Configurations

The three baseline architecture configurations A, B and C are shown in Table 4.2. A is the minimal architecture, with only a dropout layer between the input and output layers. B is the preferred baseline architecture. It includes a couple of fully connected layers, each followed by a dropout layer. This extra depth offers an improvement over the minimal approach, suggesting that the model benefits from a more complex classifier. The alternative architecture, C, is one fully connected and one dropout layer deeper than B. Although having the potential of an even more complex classifier, this architecture performed worse than B. Most likely, it is caused by overfitting as a result of the increased number of parameters. However, the performance remains stable, even after many epochs, implying that the network was somewhat overfitted before it was able to achieve a loss and accuracy on par with configuration B.

Config.	Layer #	Name	Layer Type	Units
A	0	input	InputLayer	4096
	1	drop	Dropout	
	2	output	Dense	98
B	0	input	InputLayer	4096
	1	drop1	Dropout	
	2	fc1	Dense	1024
	3	drop2	Dropout	
	4	fc2	Dense	
	5	drop3	Dropout	
	6	output	Dense	98
C	0	input	InputLayer	4096
	1	drop1	Dropout	
	2	fc1	Dense	1024
	3	drop2	Dropout	
	4	fc2	Dense	
	5	drop3	Dropout	
	6	fc3	Dense	
	7	drop4	Dropout	
	8	output	Dense	98

Table 4.2: Baseline network architecture configurations. Note that the layer type refers to the Keras 2 API. All Dense (fully connected) layers use ReLU activation functions, except the output layer of every configuration, which use softmax. All Dropout layers use a 0.5 dropout rate.

Extra Input Configurations

The three architectures with an additional expression input, with configurations D, E and F, are shown in Table 4.3. These architectures have multiple inputs and therefore the layers do not follow a fully sequential order. Rather, they each have two branches which merge together at some point. The layer numbering X-Y suggests this, where X denotes the consecutive order and Y denotes the branch it belongs to. The branches are merged in the concatenate layer by concatenating the output of the preceding layer branches. For every configuration, the concatenate layer is where the expression input is incorporated, without any layers in between. The feature input layer, however, has a varying degree layers between it and the concatenate layer, allowing the network to do some computation on the feature input alone before employing the expression data.

Of these configurations, D is the minimal, E is the preferred, and F is the alternate. Compared to the baseline configurations, D is similar to A, while E and F are similar to B. In D, the extra input is added right after the dropout layer. Networks with this architecture can then only use a single layer, the output, to make sense of the combined inputs. E allows a little more leeway, with an additional fully connected and dropout layer pair both before and after the facial expression input is injected. Its only difference to the baseline configuration B, is the additional input. The first fully connected layer allows for some processing of the feature input alone, before its output is processed in combination with the expression input in the following fully connected layers, the last of them outputting an identification classification. As with B, however, deepening the network produced no further improvement, with networks not achieving similar performance and rather showing signs of overfitting. Lastly, F inserts the expression input right after the initial dropout layer. A network with this configuration does perform better than D, but never reaches the performance levels of E, supporting the benefit of additional processing of the feature input.

Extra Output Configurations

The architecture configurations having an extra expression output are named G, H and I, and are shown in Table 4.4. Similar to the extra input architectures, these architectures have branches, and the layer numbering scheme for branches is identical. Unlike the extra input case, however, G, H and I never merge the branches. A network of this architecture would then output two separate results, one for identification and one for expression. As a result, the model would also have multiple loss and accuracy values, one for each output layer. Consequently, the visualization tool cannot be used for these networks, as it does not support multiple values for each metric. Although it is the face recognition part of the network we are truly interested in, it is unlikely that a network would gain an identification classification advantage from the enforced simultaneous classification if the expression results are underwhelming. To examine the extra output configurations, we must therefore consider the performance results for both classification tasks. The extra output networks are then supposed to do more than both the previous approaches, with no additional information.

In test architectures that tried to classify only the facial expression, validation accuracy did not rise far above 50%, suggesting that the networks had difficulty using the feature input to classify expressions. This may very well be a result of the chosen pretrained model, which was trained to recognize faces and thus has learned to prioritize features which help with identification, not expression classification. It seems that there might be some overlap between the features used to classify identification and expression, but not enough to use the pretrained model as a feature extractor in a proper expression classifier. Naturally, this lack of focus on important expression features inhibits the extra output architectures, which are tasked with classifying both the identification and expression.

Config.	Layer #	Name	Layer Type	Units
D	0-1	exp_input	InputLayer	7
	0-2	feat_input		4096
	1-2	drop	Dropout	
	2	concat	Concatenate	4103
	3	output	Dense	98
E	0-0	exp_input	InputLayer	7
	0-2	feat_input		4096
	1-2	drop1	Dropout	
	2-2	fc1	Dense	1024
	3-2	drop2	Dropout	
	4	concat	Concatenate	1031
	5	fc2	Dense	1024
	6	drop3	Dropout	
	7	output	Dense	98
F	0-1	exp_input	InputLayer	7
	0-2	feat_input		4096
	1-2	drop1	Dropout	
	2	concat	Concatenate	4103
	3	fc1	Dense	1024
	4	drop2	Dropout	
	5	fc2	Dense	
	6	drop3	Dropout	
	7	output	Dense	98

Table 4.3: Extra input network architecture configurations. Note that the layer type refers to the Keras 2 API. Architectures are not fully sequential, with some layers only appearing in certain branches. Layers in branches have X-Y layer numbers, with X referring to layer number and Y referring to branch number. All Dense (fully connected) layers use ReLU activation functions, except the output layer of every configuration, which use softmax. All Dropout layers use a 0.5 dropout rate. The Concatenate layer is used to merge branches, by concatenating the output from the branches in the preceding layers.

G is the minimal approach, H is the preferred and I is the alternate. G resembles baseline configuration A, with the distinction that the dropout layer forwards its values to two output layers instead of one. In this configuration, the input to the identification output layer and the expression output layer is not processed by any common layers that employ weights. H shares similarities with B, having two fully connected layers, separated by dropout layers, before the identification output layer. An additional fully connected layer, followed by dropout, is employed prior to the expression output layer. The two first fully connected layers are shared by the output layers and are influenced by both during training, while the `fc3` layer is only affected by the backpropagation updates from the expression output layer. We can consider the computations of the layers that are only affected by one output layer as specialized processing for the specific classification task.

The final configuration, I, is similar to H, but has identification specific processing instead of expression specific processing. The change leads to poorer overall performance compared to H, and, interestingly, the identification performance of I can be improved by removing the specialized layer. The situation is likely similar to the baseline configuration C, where the additional parameters enable overfitting. Doing the equivalent to H, on the other hand, would result in an expression performance decrease. This demonstrates the need for further and more specialized processing of the features to gain a satisfactory expression classification. This is somewhat unsurprising, regarding the lack of focus on expressions in the pretrained model, as previously mentioned. A network with specialized processing for both classifications has identification and expression performance similar to I and H, respectively. This aligns with the previous observations that specific identification processing causes harm, while specific expression processing is beneficial.

Config.	**Layer #**	**Name**	**Layer Type**	**Units**
G	0	input	InputLayer	4096
	1	drop	Dropout	
	2-1	id_output	Dense	98
	2-2	exp_output		7
H	0	input	InputLayer	4096
	1	drop1	Dropout	
	2	fc1	Dense	1024
	3	drop2	Dropout	
	4	fc2	Dense	
	5	drop3	Dropout	
	6-1	id_output	Dense	98
	6-2	fc3		1024
	7-2	drop4	Dropout	
	8-2	exp_output	Dense	7
I	0	input	InputLayer	4096
	1	drop1	Dropout	
	2	fc1	Dense	1024
	3	drop2	Dropout	
	4	fc2	Dense	
	5	drop3	Dropout	
	6-1	exp_output	Dense	7
	6-2	fc3		1024
	7-2	drop4	Dropout	
	8-2	id_output	Dense	98

Table 4.4: Extra output network architecture configurations. Note that the layer type refers to the Keras 2 API. Architectures are not fully sequential, with some layers only appearing in certain branches. Layers in branches have X-Y layer numbers, with X referring to layer number and Y referring to branch number. All Dense (fully connected) layers use ReLU activation functions, except the output layers of every configuration, which use softmax. All Dropout layers use a 0.5 dropout rate. The output of each configuration comes from the two output layers.

Chapter 5

Results

This chapter presents the results of the thesis work relevant for answering the research questions defined in Section 1.4, for both the visualization tool and the case study. We will explain the process of generating the results and elaborate on their quality and reliability, but will save the discussion of their implications for Chapter 6.

5.1 Visualization Tool

This section will focus on showcasing a representative selection of visualizations generated using the various visualization techniques implemented in the visualization tool. The visualizations were produced using the tool, but we will display them as independent images instead of incorporated into the web interface. We refer the reader to Appendix B and Appendix C for an overview of the implemented visualization tool.

5.1.1 Example Networks

The visualization results were produced using two different example ANNs. The first network is a simple CNN trained on the Modified National Institute of Standards and Technology (MNIST) dataset[1], which is commonly used as a toy problem. This network is quickly trained from scratch to near perfect accuracy, which allowed us to easily show how the visualizations evolve throughout the training progress. The second network is the more advanced VGG network, more specifically the 16-layer version in configuration D [47]. In this case, we used a model with weights pretrained on ILSVRC 2012 image set [45]. Some of the visualizations, such as those from the deconvolutional network technique, require a large network with many convolutional layers to demonstrate their full capabilities. Using a fully trained VGG model, we were able to present a second set of visualizations illustrating more complex features, without having to spend a large amount

[1]http://yann.lecun.com/exdb/mnist/

of time training the network. The VGG network also allowed us to display how the visualizations handle RGB images, compared to the grayscale images of the MNIST dataset. Both networks were implemented following examples provided by Keras, and in the following sections we describe their implementations and training processes in detail.

Note that there are several ways to count the number of layers in an ANN. Some conventions do not include pooling or flattening layers, or even the input and output layers. To ensure consistency when describing the network architectures, we will define the number of layers as the number of Keras layers, as listed in their documentation [51].

The MNIST Network

The MNIST network is a simple CNN with three convolutional layers, including the max pooling layer. It is trained on the MNIST dataset, which contains 60,000 28x28 grayscale images of handwritten digits. The network architecture can be seen in Figure 5.3, and was created following an example implementation from Keras [2]. The network consists of 10 layers, where three are convolutional and pooling layers, and two are fully connected, denoted as dense layers in Keras. In addition to an input layer, there are also two dropout layers to prevent overfitting, and a flattening layer to convert the 3D output from the convolutional part to 1D input for the fully connected part. The model is trained using an AdaDelta optimizer with default Keras values[3] to minimize the categorical cross-entropy loss.

To generate the results, the network was trained for two epochs with a batch size of 128, shuffling the data at each epoch. In addition to the 60,000 training images, 10,000 images were used as validation data. The 10 images in Figure 5.1 were extracted from the validation dataset to be used as visualization input images. The training process was repeated using each of these images as input for the visualizations. The visualizations were produced five times during each epoch, in addition to once at the very beginning of the training, giving a total of 11 different sets of visualization results for each of the 10 digits. The produced visualizations were manually processed and a representative selection were extracted, presented in Section 5.1.2.

The VGG Network

The VGG network is a significantly deeper CNN than the first network. It consists of five blocks of 3-4 convolutional layers each, including the max pooling layers, followed by a flattening layer and three fully connected layers, for a total of 23 layers. The full architecture can be seen in Figure 5.4. The VGG model used has weights trained on the ILSVRC image set and is available through the Keras Applications module[4][5].

[2]https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py
[3]https://keras.io/optimizers/#adadelta
[4]https://keras.io/applications/#vgg16
[5]https://github.com/fchollet/keras/blob/master/keras/applications/vgg16.py

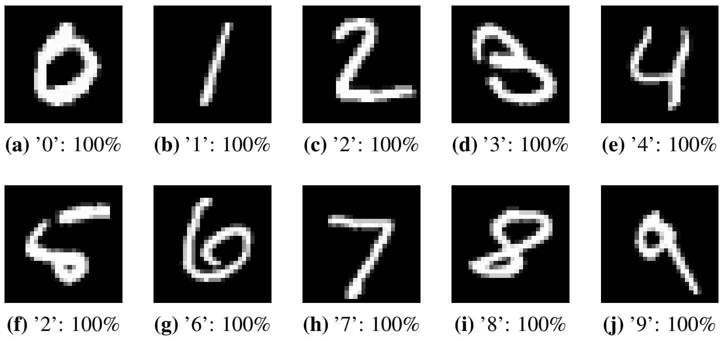


Figure 5.1: The images used as visualization input for the MNIST network, captioned with their corresponding classification result and confidence at stage 1 of training. Note that **f** was incorrectly classified as '2'. At stage 2, all images were correctly classified with 100% certainty.

As mentioned, the VGG network employed is a fully trained model, meaning that the results generated using this network only consist of visualizations produced at the very end of the training process. The three visualization input images seen in Figure 5.2 were used to produce visualizations, with the image in 5.2a being the main visualization input image. The other two images were used to produce visualizations to compare to. They were selected as they are both related to the main image in some way. Specifically, one contains a bicycle similar to the tricycle in the main image, and the other contains a woman with similar features to the girl in the main image. The visualization techniques were repeated a number of times with various configurations and the different input images. The resulting visualizations were manually processed, and a representative selection are presented in Section 5.1.3.

5.1.2 MNIST Visualization Results

This section presents the visualization results produced by the various visualization techniques for the MNIST network.

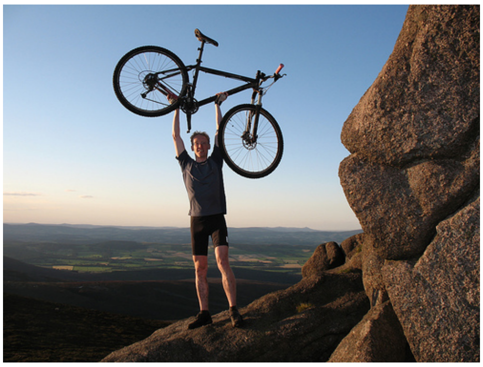
Training Progress

The visualizations of the training progress can be seen in Figure 5.5, showing the batch accuracy and batch loss. To produce these visualizations, the training was expanded from two epochs to six, to show a longer graph. The blue lines display the training accuracy and loss, while the green dots display the validation accuracy and loss at each epoch.

Layer Activations

Figure 5.6 shows the fully trained network's layer activations for all layers, except the flattening and dropout layers. Flattening layers are excluded because they do not provide any extra information, and are generally too large to be presented in a readable manner.

(a) 'tricycle, trike, velocipede': 99.95%


(b) 'cliff, drop, drop-off': 64.35%

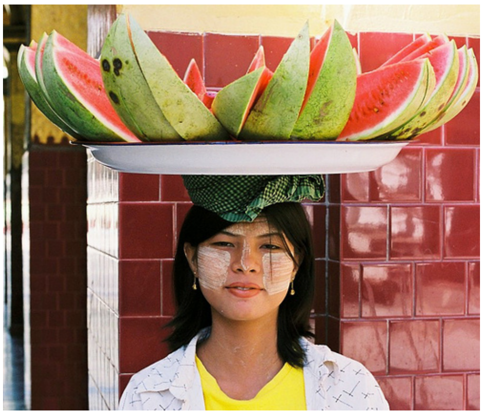

(c) 'sombrero': 50.91%

Figure 5.2: The images used as visualization input for the VGG network, captioned with their corresponding classification result and confidence.

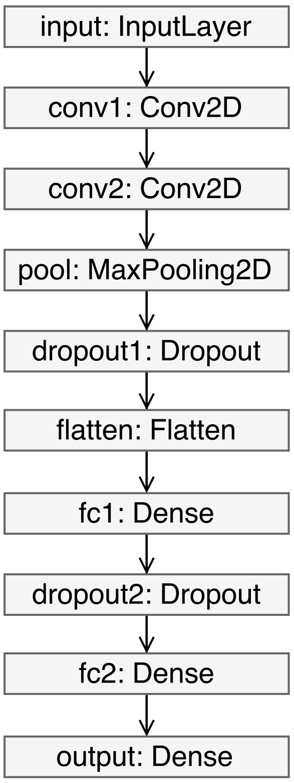


Figure 5.3: Architecture of the MNIST network with layer names and types. Note that the layer type refers to the Keras 2 API.

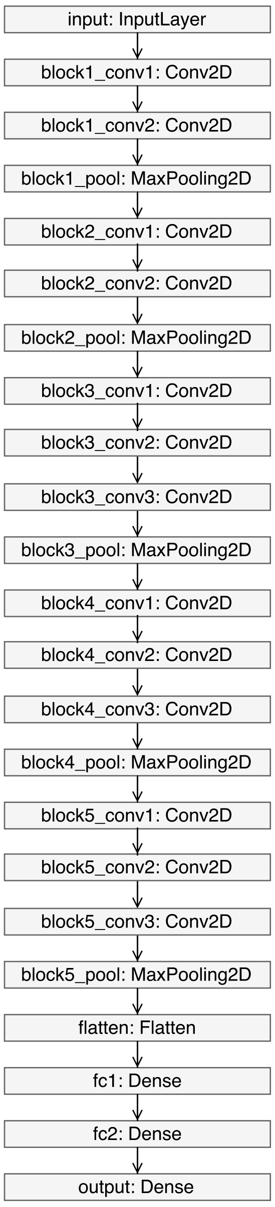
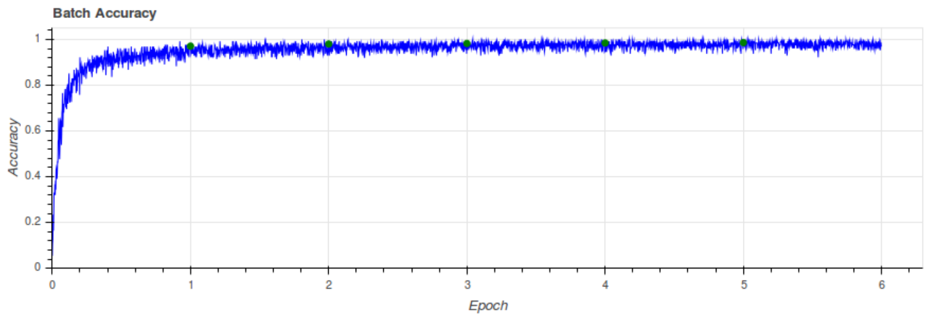


Figure 5.4: Architecture of the VGG network with layer names and types. Note that the layers refer to the Keras 2 API.



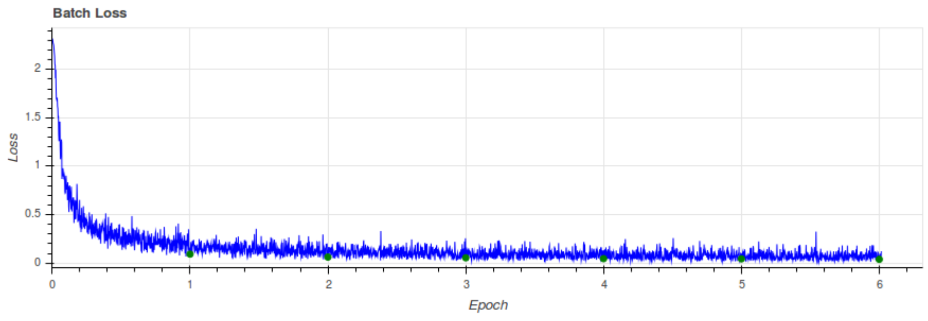


Figure 5.5: Batch accuracy and loss over epoch for the MNIST network. Validation accuracy and loss displayed as green dots at each epoch. Note that the validation accuracy and loss are best viewed electronically.

Dropout layers are excluded since they are identical to the preceding layer. The feature maps of the convolutional layers are displayed in a grid so that they can be viewed together all at once. The fully connected layers are displayed as pixels laid out horizontally, scaled to fit the width of the page.

Figure 5.7 illustrates the transition of the layer activations throughout the training process. It is illustrated by the `conv2` layer activations, as they show a progression that is easy to understand and interpret. The activations are shown at three stages: the first stage at the beginning of training, a middle stage halfway in the training, and a final stage at the end of training.

Saliency Maps

Figure 5.8 shows saliency maps at various stages of the training process for each of the visualization input images classified in Figure 5.1. Note that stage 0 is an initial stage before the training has started. The illustration includes more of the earlier stages than later stages because most of the changes to the saliency map occur at the beginning of the training process.

Deconvolutional Network

For the deconvolutional network technique, visualizations of all 32 feature maps in the `pool` layer were produced. Four feature maps that illustrate different behaviours were manually selected and are displayed together with the visualization input image in Figure 5.9.

As mentioned, the MNIST network is not optimal for demonstrating the deconvolutional network technique, since it consists of only three convolutional layers. The input images are composed of simple lines, and the features detected in the convolutional layers are mainly edges. The capabilities of a deconvolutional network is better illustrated using the VGG network.

Deep Visualization

The deep visualizations were produced with a learning rate of 200 for 50 iterations. The L_2 decay, blur interval and blur standard deviation were set to 0.0001, 4 and 1 respectively, matching one of the useful combinations in Table 2.1. The deep visualizations of the output layer are shown in Figure 5.10. The visualizations from stages 1, 5 and 10 are presented, to show how they evolve during training. When producing several images with deep visualization, the results will vary. This variation reveal how invariant the network is, by demonstrating how its units maximally activate on similar, but not identical, images. Deep visualizations of some of the lower layers, specifically `fc2` and `fc1`, are presented in Figure 5.11 and Figure 5.12, respectively. These show a selection of visualizations produced at the final stage of training.

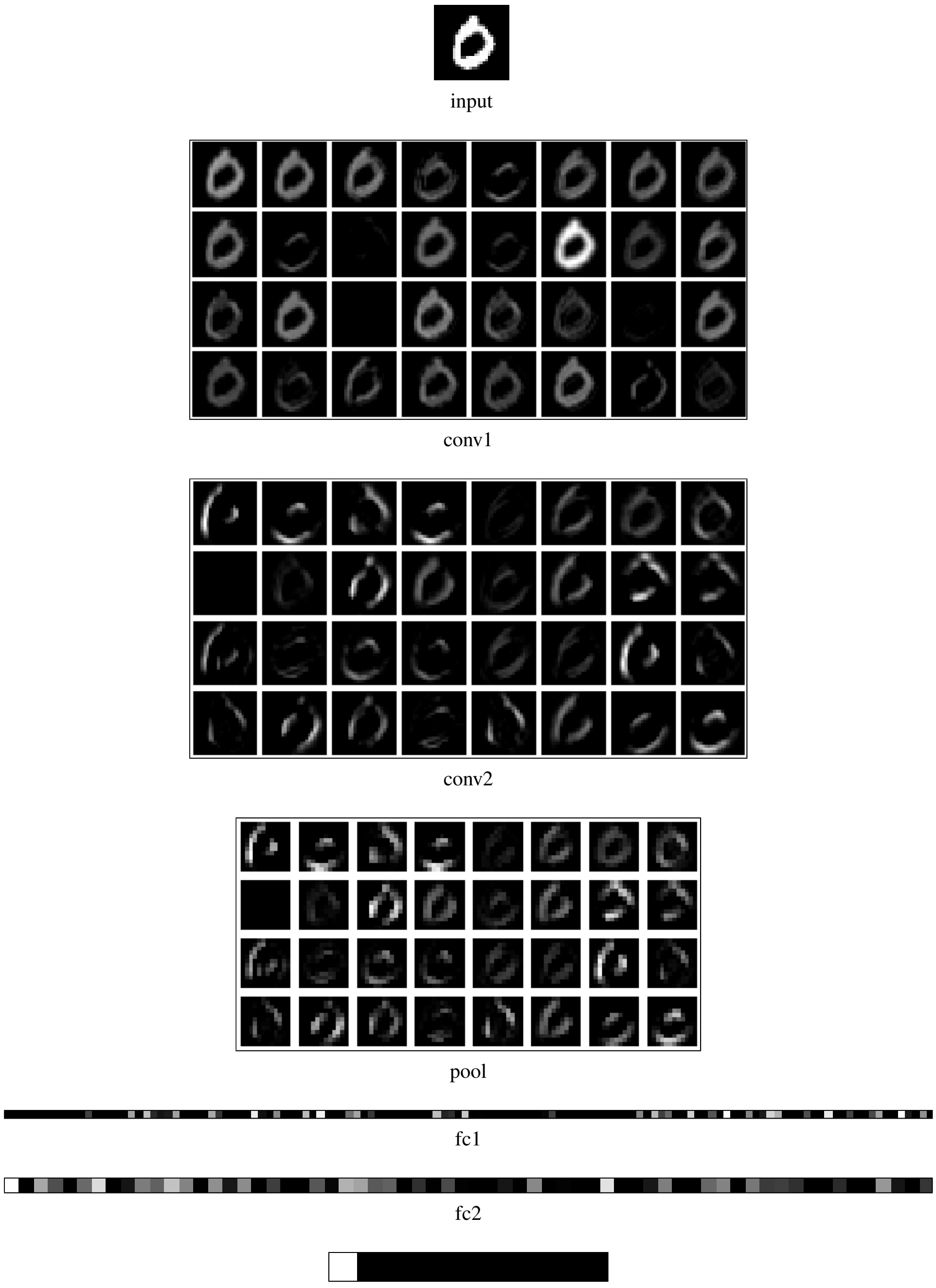
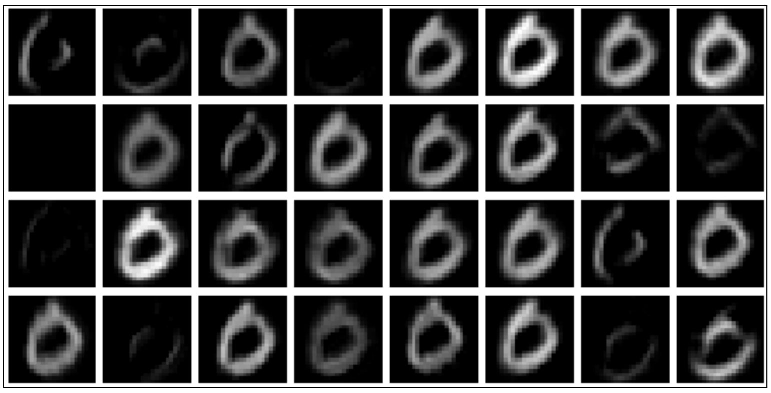
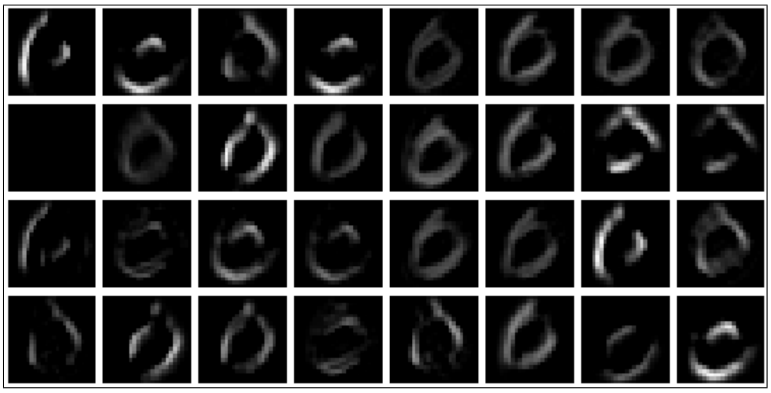


Figure 5.6: Layer activations for all layers of the MNIST network, excluding flattening and dropout layers, for visualization input image 0 after two epochs.


Stage 1


Stage 5

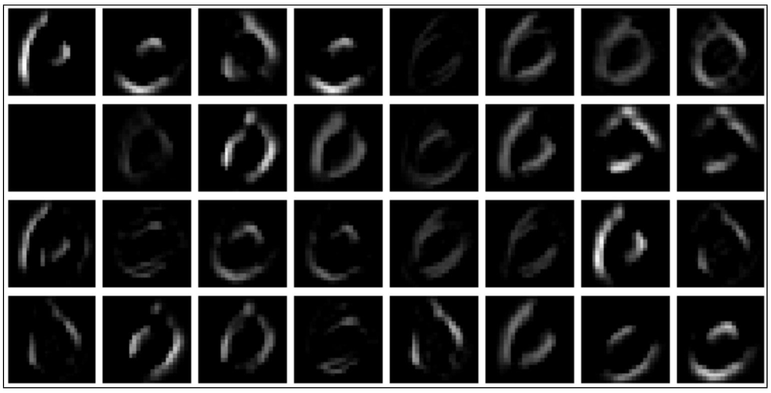

Stage 10

Figure 5.7: Layer activations for the `conv2` layer of the MNIST network at various stages in the training process.

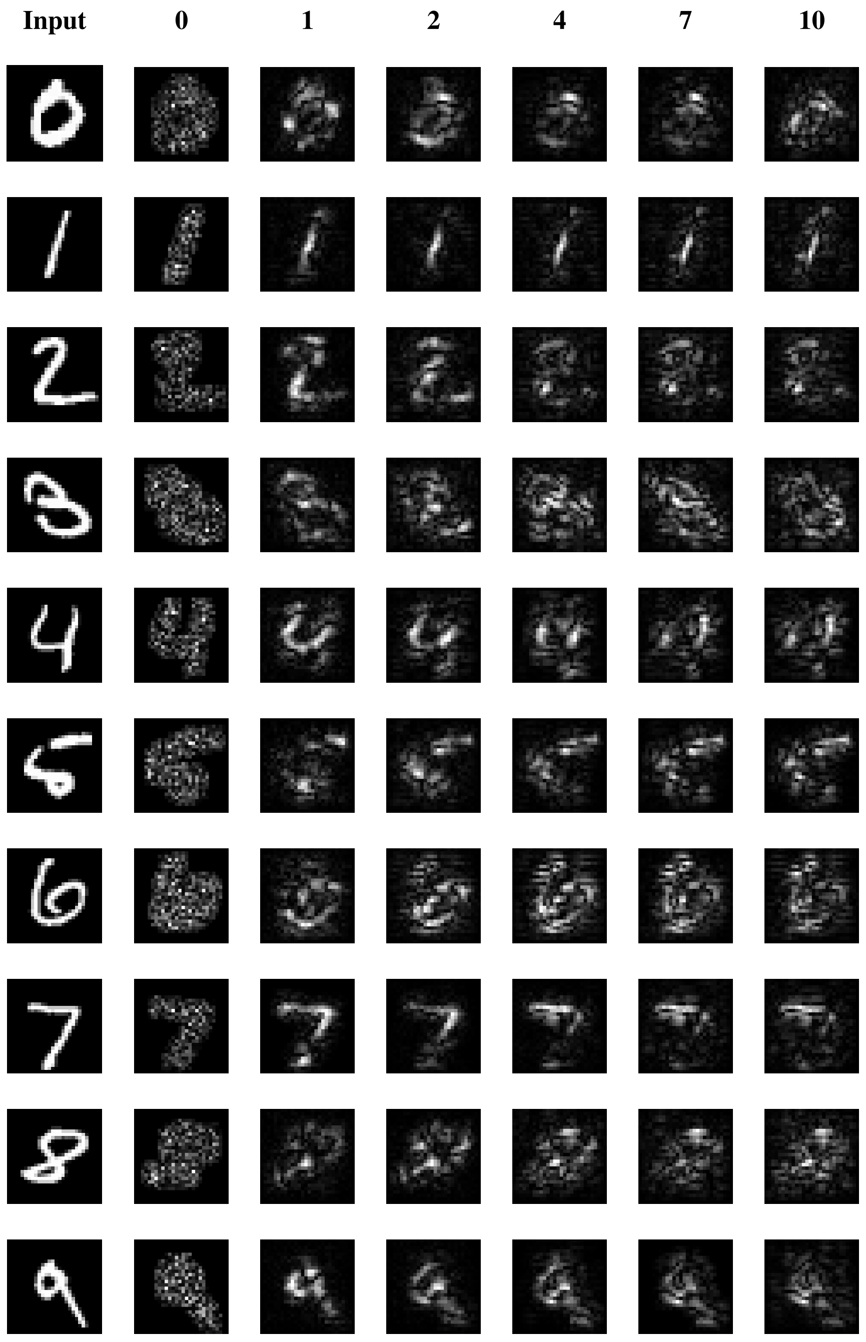


Figure 5.8: Saliency maps for the MNIST network with visualization input images 0-9 classified as presented in Figure 5.1. Input images are displayed in the first column, followed by samples collected at certain stages of training. The two training epochs consisted of ten evenly distributed stages. Note that stage 0 is produced before the training has started.

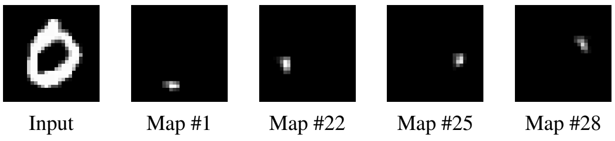


Figure 5.9: Visualizations produced using a deconvolutional network for the visualization input image in **a**. A selection of four different feature maps from the `pool` layer is displayed.

5.1.3 VGG Visualization Results

This section presents the visualization results produced by the various visualization techniques for the VGG network. Note that there was no training completed while generating the visualizations. Naturally, there does not exist any training progress to visualize.

Layer Activations

Figure 5.13 show the layer activations for the `block1_conv2` layer of the VGG network. Although the network contains many layers, we only display the single example to avoid cluttering. The `block1_conv2` was chosen because it has a varied set of feature maps. Additionally, the amount of feature maps is manageable.

Saliency Maps

Figure 5.14 shows the saliency maps of the three visualization input images as classified in Figure 5.2.

Deconvolutional Network

The deconvolutional network technique is illustrated in a number of different figures. Figure 5.15 shows visualizations of six different feature maps of the `block5_pool` layer produced using a deconvolutional network autogenerated for the VGG network. The feature maps are selected from the top 20 maximally activated feature maps, and are chosen as to show the activations of different regions of the visualization input image. In Figure 5.16, two of these feature maps are compared to the same feature maps visualized using two different visualization input images.

To illustrate the deconvolutional technique employed on lower layers, Figure 5.17 shows a comparison of feature maps in layer `block3_pool`, for two separate visualization input images. Figure 5.18 shows visualizations of feature maps in the even lower `block1_pool` layer, using the image in 5.2a as input for the deconvolutional network. Note that all visualizations of lower layers are cropped to improve readability, and that they originally appeared at different parts of the image.

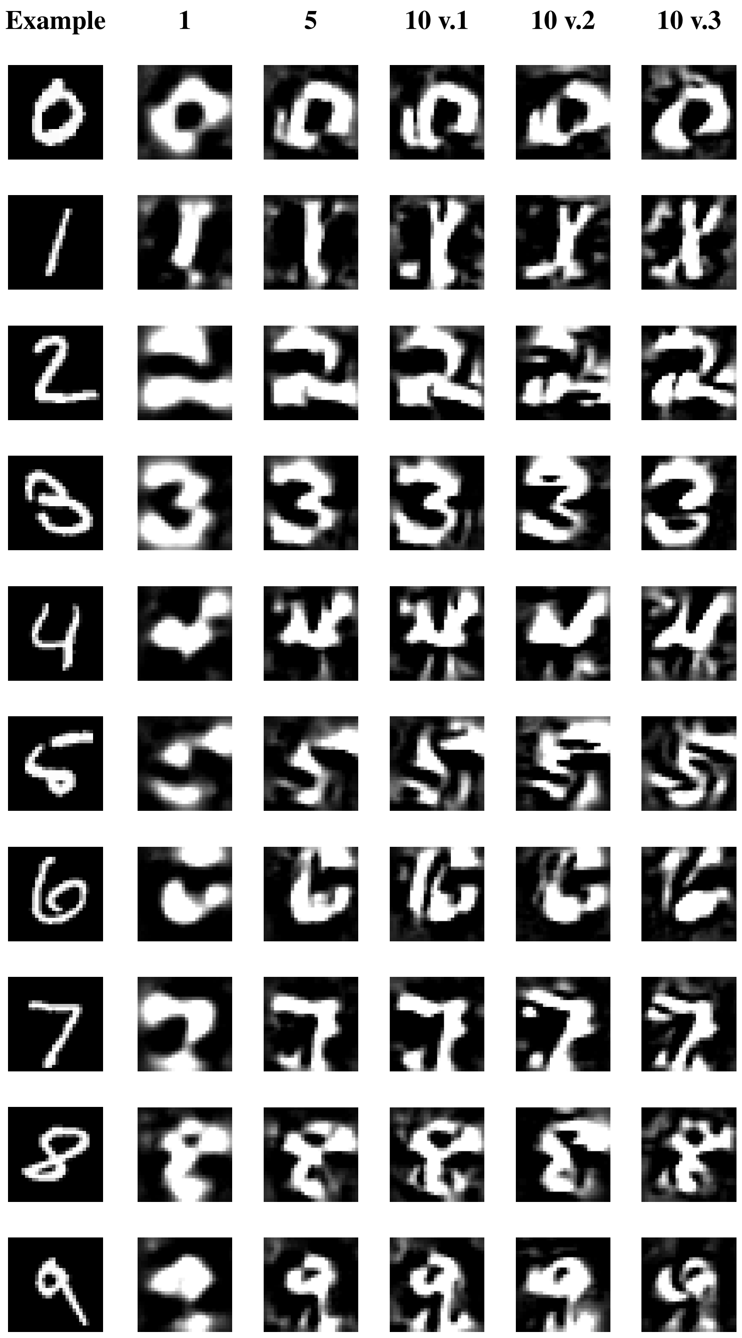


Figure 5.10: Deep visualization of the output layer for all 10 output classes of the MNIST network. An example image from each class is shown in the first column. Column 2-4 shows samples collected at certain stages of training. The two training epochs consisted of ten evenly distributed stages. The last three columns show the variations acquired from multiple deep visualization from the same unit.

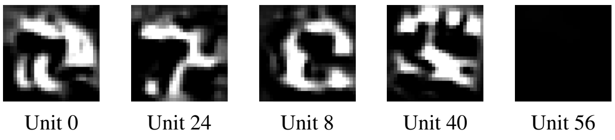


Figure 5.11: Deep visualizations for a selection of the units in the `fc2` layer of the MNIST network.

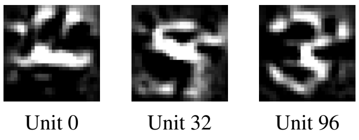


Figure 5.12: Deep visualizations for a selection of the units in the `fc1` layer of the MNIST network.

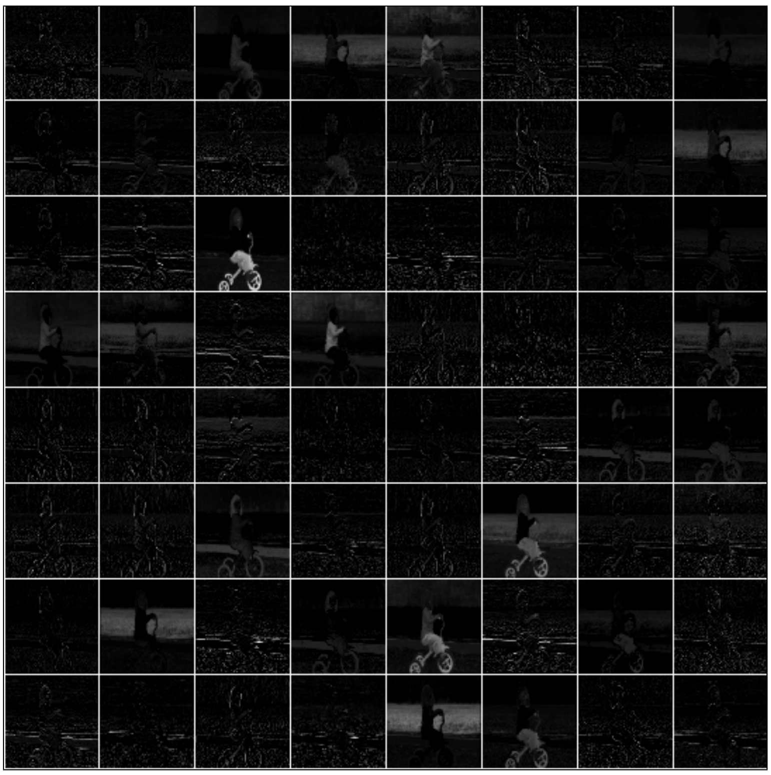


Figure 5.13: Layer activations for the `block1_conv2` layer of the VGG network.

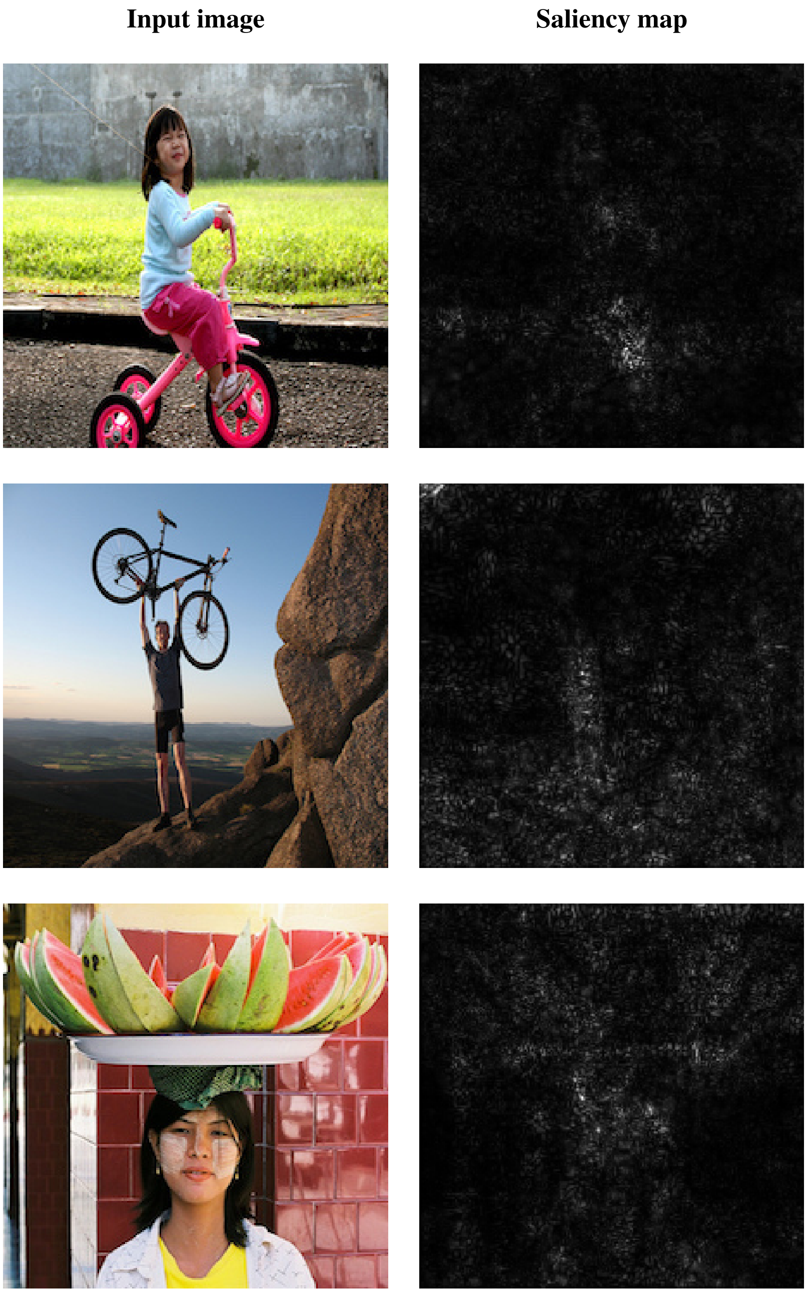
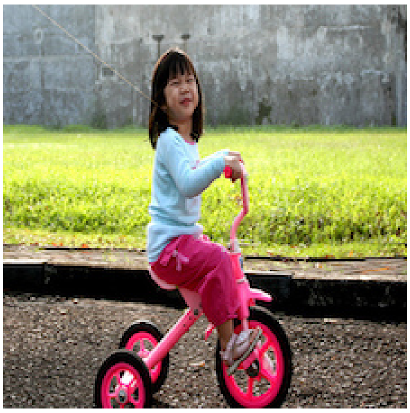
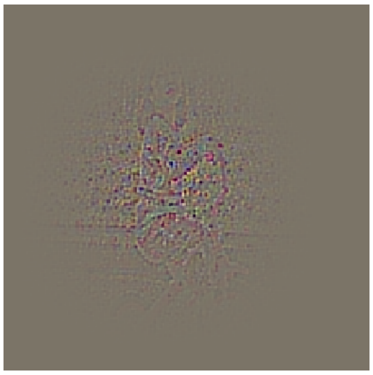
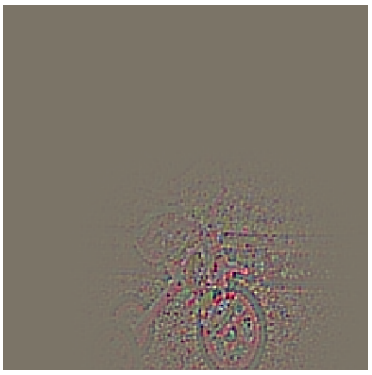
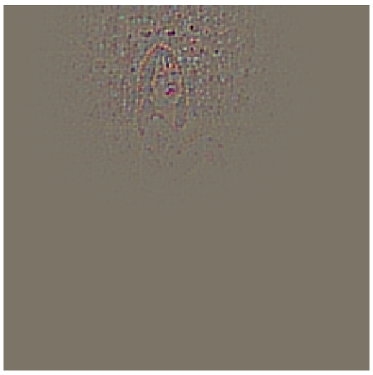


Figure 5.14: Saliency maps for the VGG network for three separate visualization input images classified as presented in Figure 5.2.


Input image


Feature map 108


Feature map 348


Feature map 380

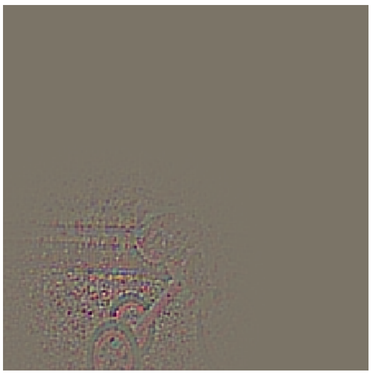
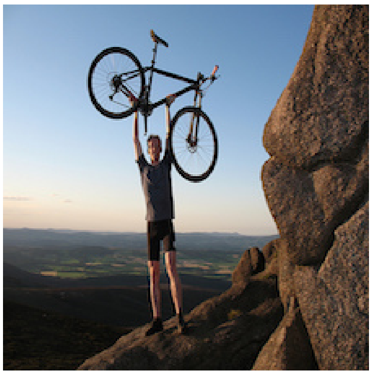
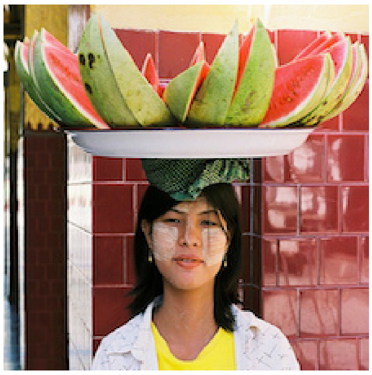
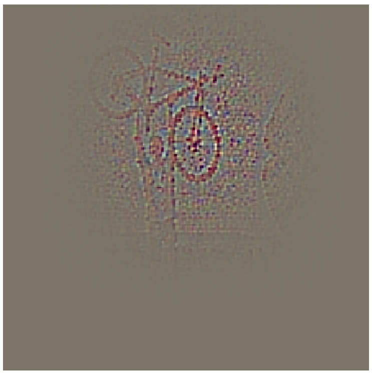
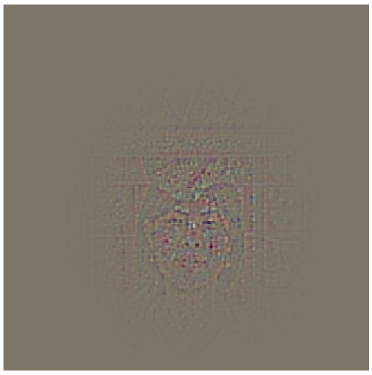

Feature map 450

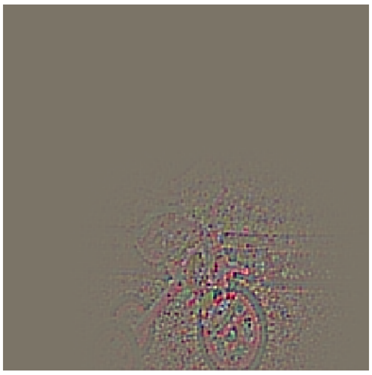
Figure 5.15: Visualizations produced using a deconvolutional network for the `block5_pool` layer. A selection of four different features from the top 20 maximally activated feature maps from the `block5_pool` layer are displayed. Note that the visualizations are best viewed electronically.


Input image


Input image


Feature map 348


Feature map 380


Feature map 348 in Figure 5.15

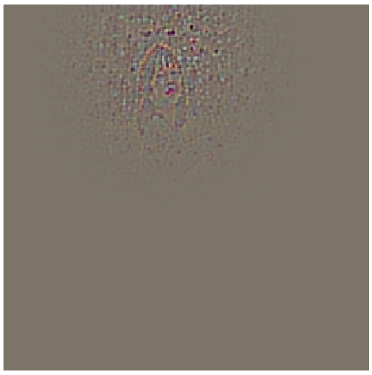

Feature map 380 in Figure 5.15

Figure 5.16: Comparison of visualizations produced using a deconvolutional network for the visualization input images in the top row, for the `block5_pool` layer of the VGG network. The middle row shows their corresponding visualizations for a specific feature map. The bottom row shows the visualizations of the same feature map for the visualization input image in Figure 5.15. Note that the visualizations are best viewed electronically.

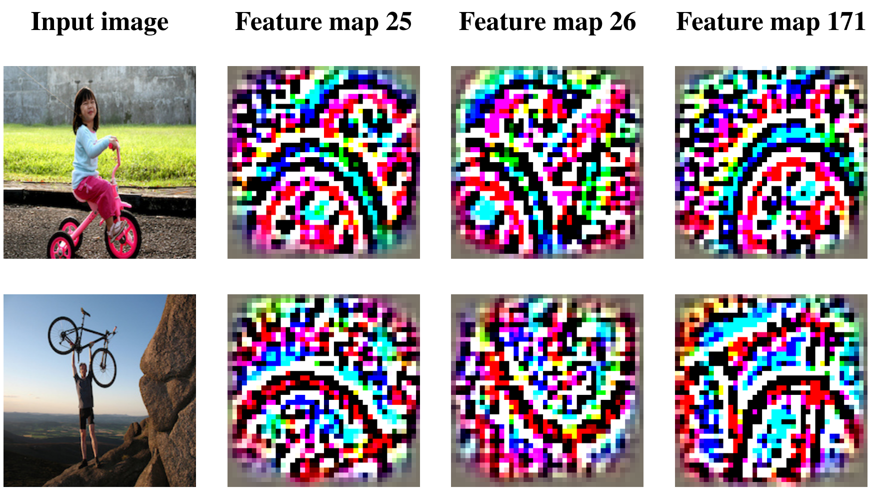


Figure 5.17: Comparison of visualizations produced using a deconvolutional network for the visualization input images in the leftmost column, for the `block4_pool` layer of the VGG network. The next columns show their corresponding visualizations for specific feature maps selected from the top 20 maximally activated feature maps for the top left visualization input image. Note that the visualizations are cropped.

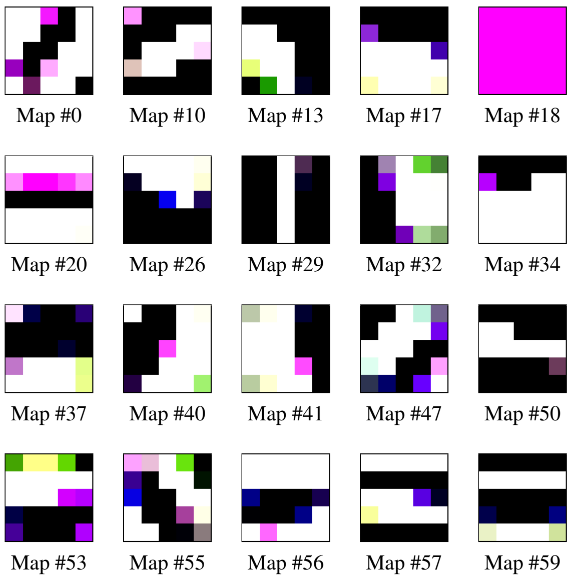


Figure 5.18: Visualizations produced using a deconvolutional network for the visualization input image in 5.2a, for the `block1_pool` layer of the VGG network. All top 20 maximally activated feature maps are displayed. Note that the visualizations are cropped.

Deep Visualization

The deep visualization technique requires a number of hyperparameters that impact the resulting visualizations. Some of the figures illustrates these impacts and thus a number of visualizations are produced using different hyperparameters settings. However, most of the deep visualizations were produced with a learning rate of 2500 for 500 iterations. Unless otherwise specified, the L_2 decay, blur interval and blur standard deviation were set to 0.0001, 4 and 1 respectively, matching the third row of Table 2.1.

Figure 5.19 presents the deep visualizations of nine different output classes of the VGG network, selected to showcase a variety of features. Figure 5.20 illustrates the variation of the deep visualization technique in producing two visualizations for the same output class with the exact same hyperparameter settings. In Figure 5.21, we present deep visualizations using each of the four useful combinations in Table 2.1, illustrating how these affect the visualization output.

Some examples of deep visualization of the `fc2` and `fc1` layers are shown in Figure 5.22 and Figure 5.23, respectively. In Figure 5.24, deep visualizations of a randomly selected unit in each pooling layer of the blocks 2-5 are presented. Since using the same learning rate and number of iterations for the lower layers as for the higher layers may not be beneficial, Figure 5.25 show how lowering these hyperparameters affect the resulting visualization. Finally, Figure 5.26 present deep visualizations of the `block1_pool` layer.

5.2 Case Study in Face Recognition

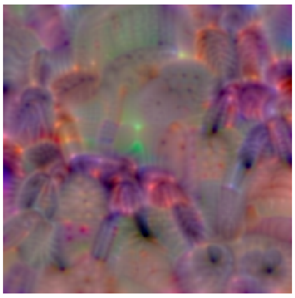
In this section we will present the performance of multiple case study networks, based on the architectures described in Section 4.2.6. We also explain how these networks were obtained, by detailing their training scheme.

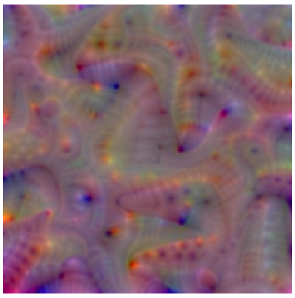
5.2.1 Network Performance

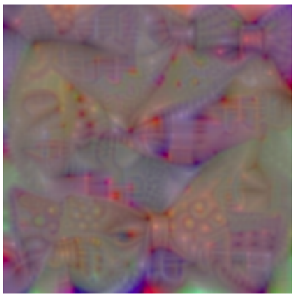
For every architecture configuration described in Section 4.2.6, a network was created, trained and tested. The performance results of each network with a single output is displayed in Table 5.1, with loss and accuracy for the validation and test sets. For the extra output architectures, the identification and expression classification performance are measured separately, and are similarly displayed in Table 5.2 and Table 5.3. The values represent the average result each network achieved on the respective sets. The differences found in the performance measurements may be modest, but considering how small the architectural differences are, they are not negligible.

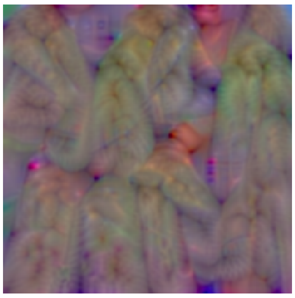
5.2.2 Training

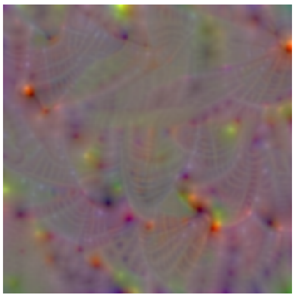
The case study networks are regarded as separate from the pretrained network, as explained in Section 4.2.6. They are also trained accordingly, using feature input instead of image input. The networks were implemented using Keras, with TensorFlow as the

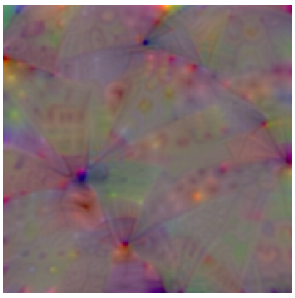

Class 76 - tarantula

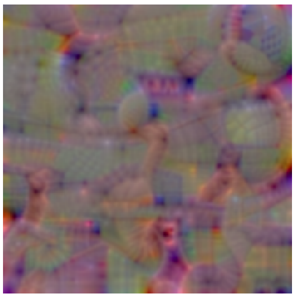

Class 327 - starfish, sea star

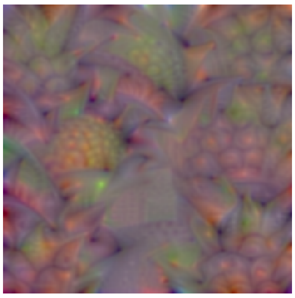

Class 457 - bow tie, bow-tie, bowtie


Class 568 - fur coat


Class 815 - spider web, spider's web


Class 879 - umbrella


Class 890 - volleyball


Class 953 - pineapple, ananas

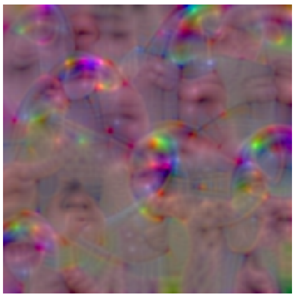

Class 971 - bubble

Figure 5.19: Deep visualization of the `output` layer for a selection of nine output classes of the VGG network. The visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1. Note that the visualizations are best viewed electronically.

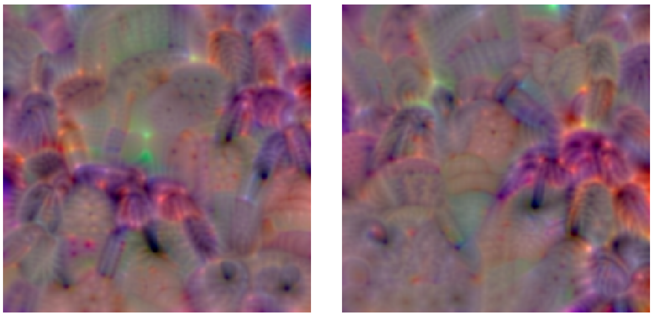


Figure 5.20: Two versions of deep visualization for the `output` layer of the VGG network for the 'tarantula' output class. Both visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1. Note that the visualizations are best viewed electronically.

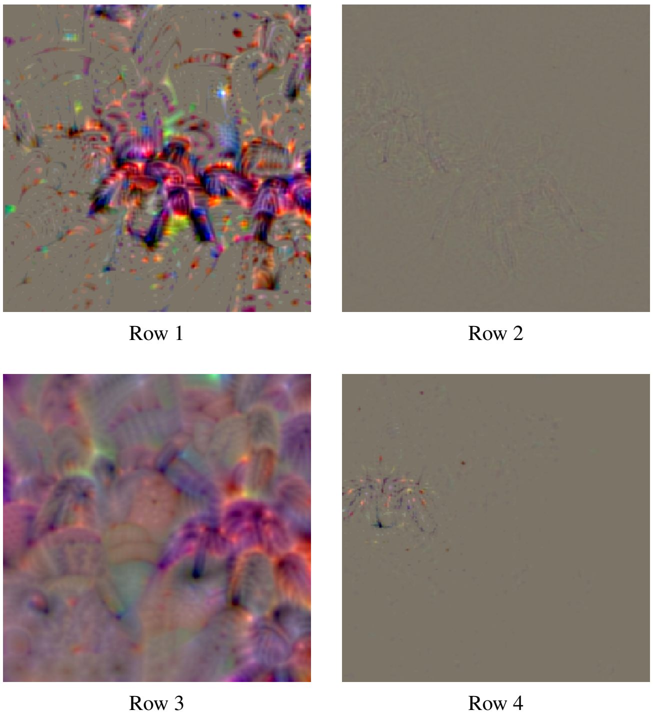


Figure 5.21: Four versions of deep visualization of the `output` layer of the VGG network for the 'tarantula' output class. All visualizations are created using a learning rate of 2500 for 1000 iterations, but with hyperparameter corresponding to each of the rows in Table 2.1. Note that the visualizations are best viewed electronically.

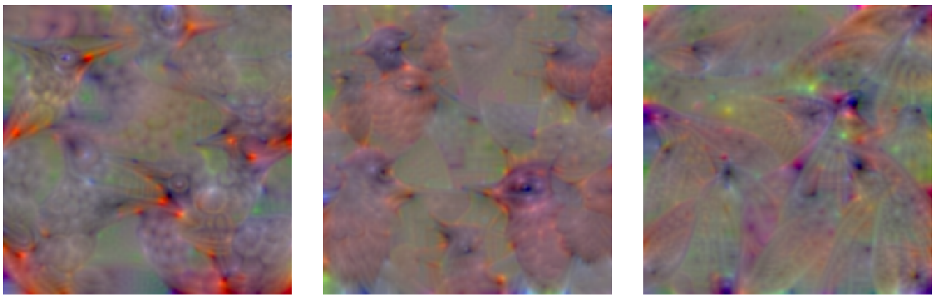


Figure 5.22: Deep visualization of three units in the `fc2` layer of the VGG network. The visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1. Note that the visualizations are best viewed electronically.

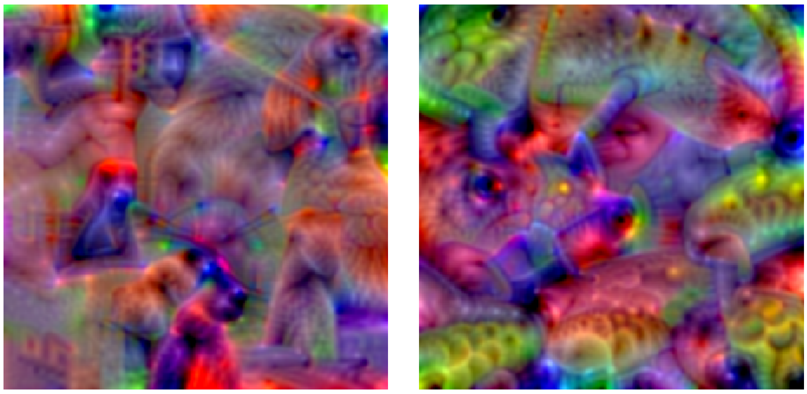


Figure 5.23: Deep visualization of two units in the `fc1` layer of the VGG network. The visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1. Note that the visualizations are best viewed electronically.

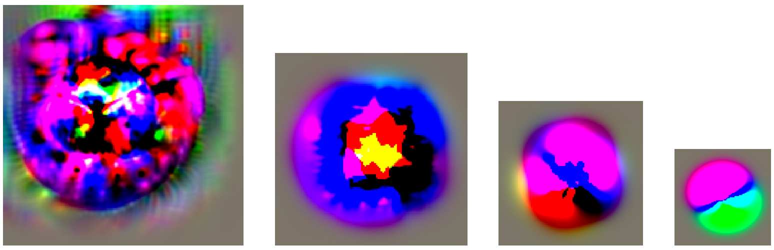


Figure 5.24: Deep visualization of a unit from each of the pooling layers in block 2-5 of the VGG network. The visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1.

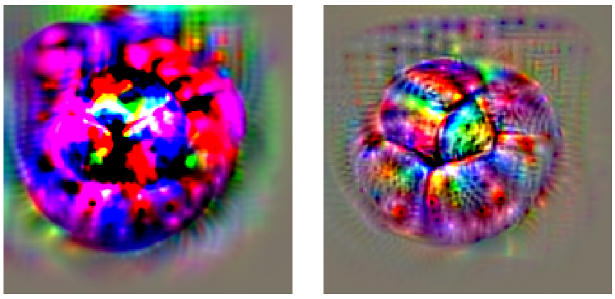


Figure 5.25: Two versions of deep visualization of a unit in the `block5_pool` layer of the VGG network. The visualizations are created using different learning rates (lr) and number of iterations. The hyperparameter settings correspond to the third row of Table 2.1.

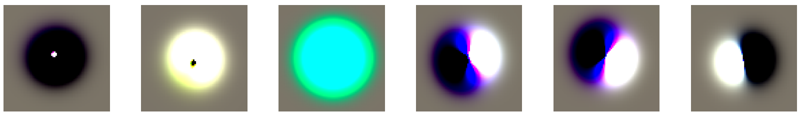


Figure 5.26: Deep visualization of six units in the `block1_pool` layer of the VGG network, with the unit index captioned below each image. The visualizations are created using a learning rate of 2500 for 1000 iterations. The hyperparameter settings correspond to the third row of Table 2.1.

Config.	**Accuracy**		**Loss**	
	Validation	**Test**	**Validation**	**Test**
A	0.77235	0.75607	0.90662	0.94242
B	0.78332	0.77866	0.84432	0.89128
C	0.77401	0.76803	0.91538	0.96147
D	0.78082	0.76703	0.87629	0.91768
E	0.80243	0.79030	0.77779	0.83946
F	0.79030	0.78232	0.82492	0.86545

Table 5.1: Performance of case study networks with a single output. Loss and accuracy measurements of both the validation set and test set are reported. Networks were trained as described in Section 5.2.2

Config.	**Identification Performance**			
	Accuracy		**Loss**	
	Validation	**Test**	**Validation**	**Test**
G	0.77252	0.75939	0.90573	0.94598
H	0.78780	0.77966	0.84904	0.89195
I	0.77052	0.76338	0.94080	0.96959

Table 5.2: Identification performance of extra output case study networks. Loss and accuracy measurements of both the validation set and test set are reported. Combined metric performance can be seen in Table 5.1. Networks were trained as described in Section 5.2.2

Config.	**Expression Performance**			
	Accuracy		**Loss**	
	Validation	**Test**	**Validation**	**Test**
G	0.48272	0.48455	1.40757	1.40566
H	0.52509	0.53407	1.30467	1.28520
I	0.52210	0.53938	1.32465	1.29725

Table 5.3: Expression performance of extra output case study networks. Loss and accuracy measurements of both the validation set and test set are reported. Combined metric performance can be seen in Table 5.1. Networks were trained as described in Section 5.2.2

backend framework. Because of small amount of weighted layers in these architectures, the training of each one is completed quickly. When training the different networks, we varied the amount of epochs used, while the batch size was fixed at 512. For the configuration A, D, and G, the networks were trained for 25, 30, and 25 epochs, respectively. The networks based on the other configurations were all trained for 50 epochs. Generally, the shorter epoch amounts were either chosen because of a lack of significant progress in the later epochs, or to prevent overfitting. The training times can be seen in Table 5.4. To generate the features on which the networks were trained, the pretrained model processed all images in the chosen dataset. Upon processing, each image was resized to have a height and width of 224 pixels, and had the mean values of the original training set subtracted from its pixel values. For the training, validation and test sets, the feature extraction took a total of 181 minutes.

Every network was trained using the adaptive learning algorithm Adam, with an initial learning rate of 0.0005. This learning rate is lower than the recommended default value, but is chosen due to the use of transfer learning. The other parameters of Adam use the default values[6]. The loss trying to be minimized by this learning algorithm was measured using categorical cross-entropy, and the weights were initialized using a Glorot uniform initializer, also known as a Xavier uniform initializer [52]. All training occurred on the same hardware, an Intel® Core™ i7-6700 3.40 GHz CPU.

Config.	A	B	C	D	E	F	G	H	I
Time	2	8	9	2	8	8	1	11	11

Table 5.4: Configuration training times. Training time, given in minutes, of each configuration network used to produce results in Table 5.1.

[6]https://keras.io/optimizers/#adam

Chapter 6

Discussion

This chapter will discuss and evaluate the results presented in Chapter 5, and reflect upon the findings.

6.1 Visualization Tool

In this section we start with interpreting and explaining each of the visualizations presented in Section 5.1.2 and Section 5.1.3. The visualizations of both the MNIST network and the VGG network will be examined for each visualization technique. The section ends with a discussion on the usefulness of the implemented visualization tool.

6.1.1 Training Progress

The only produced visualizations of the training progress were the batch accuracy and batch loss plots of the MNIST network in Figure 5.5. Both plots indicate a healthy network, reaching a training accuracy of close to 100% shortly after three epochs of training. Even though the training continued for another three epochs, the network shows no signs of overfitting as the validation accuracy is still high.

6.1.2 Layer Activations

In Figure 5.6, we see the layer activations for most layers of the MNIST network. The first convolutional layer does not seem to detect particularly interesting features, but the second convolutional layer apparently detects edges in various locations, which is typical for the first couple of layers of a CNN. The black feature maps indicate that their corresponding filters respond to features that are not found in the selected visualization input image. If the feature map remains black for several input images, it may indicate a dead filter, possibly caused by too high a learning rate. The rest of the layer activations present an overview of the network behavior resulting in a correct classification of the image with

100% certainty, seen as a single bright activation in the output layer.

Figure 5.7 shows how the layer activations develop as the training progresses. At stage 1, the convolutional layer has not yet learned to identify features. We see a significant improvement in stage 5, where the edges of the visualization input image emerge. From stage 5 to stage 10, there are no major improvements, as the network has stabilized.

In Figure 5.13, we see the activations in the second convolutional layer of the VGG network. By studying the feature maps and comparing them to the visualization input image, we can conclude that they do not only detect edges, but also other features, like color. For instance, the third feature map on the third row seems to activate greatly on the color pink, and some of the other feature maps detect the green color of the grass.

6.1.3 Saliency Maps

The evolution of the saliency maps while training the MNIST network is shown in Figure 5.8. We see that at stage 0, which is before any training has been conducted, the saliency appears to be randomly computed, except for the black areas, where the gradients are zero. The random computation of the saliency is coherent with the random prediction of the input image as the network has not seen any images yet. Looking at the following stages, it is evident that the most prominent changes happen during the first stages of training. From stage 7 to stage 10, there are only small adjustments present. The reason for this is that the loss stabilizes after some time. Recall that the saliency is the gradient of the loss with respect to the visualization input image. Thus, as the loss stabilizes, so will the gradient and, consequently, the saliency.

As seen in Figure 5.1, all visualization input images were correctly classified at stage 1, except for digit 5. Looking at the stage 1 saliency map of digit 5, we see two highlighted spots at the top right and lower middle. These are the parts that the network deems most important when classifying the visualization input image. In this case, the image was incorrectly classified as ’2’. We can see that the location of the highlighted areas are somewhat similar to those of the stage 1 saliency map for digit 2. The network has learned a representation for ’2’ that it finds similar to digit 5, and it classifies accordingly. Note that the visualization input image for 5 is a particularly poorly written example. These kinds of images are necessary to fully generalize the network, but at an early stage of training, the network has not seen enough images to tackle them. At stage 2, the network has already improved and is able to correctly classify the digit 5.

Another interesting observation is that for many of the visualization input images, the first couple of stages resembles the image the most. For instance, for digit 3, the saliency map of stage 1 tracks the image fairly well, while stage 10 presents a significantly more noisy and confusing saliency map. As the training progresses, the network sees more and more images and thus it generalizes. The final saliency maps show us that the network could be looking mostly for certain characteristics instead of the whole picture when classifying the digits. This is especially evident for digits 1, 4 and 7. For digit 1, we see that its last couple of saliency maps does not highlight the digit from the very bottom to the top, but focuses on the middle part of the line. This could be because the 1s that the network has seen vary in length, and thus the middle is the common and most important part influencing the classification. Similarly, the final saliency maps of digit 4 focus on the two vertical top lines, while the final saliency maps of digit 7 focus on the top horizontal line and practically ignores the diagonal line. These are characteristics that distinguish them from the other digits, and consequently helps the network in its classification.

In addition to the saliency maps for the MNIST network, three saliency maps are also presented for the VGG network, in Figure 5.14. The visualization input images of VGG are far more complex than those of MNIST, which is evident in their produced saliency maps. It is hard to see obvious shapes in the saliency maps, but they still indicate what areas of the images were the most important. The top image was classified as a tricycle with 99.95% certainty. Its saliency map shows us that the focus was on the little girl and especially how her legs are positioned. This tells us that the network considers the specific pose as an indication of a tricycle. Most likely, many of the images it has seen of a tricycle have included a child sitting on it. The saliency map also slightly highlights the curb, which may be another hint towards a tricycle. The middle image was classified as a cliff with 64.35% certainty. Since the image also contains a bicycle and a man, it is difficult for the network to know which object to focus on. In the corresponding saliency map, we see that the man is highlighted along with parts of the rock and background. This can be expected, as the network has most likely been exposed to many cliff-labeled images in training which contained people posing for a photo, as well as a scenic backdrop. Naturally, the texture of the rock is also important for the classification. Since the saliency map is produced based on the chosen classification, the bicycle is not highlighted at all. A possible reason for the lack of a bicycle classification might be its unusual position, both in relation to the man and the background. The bottom image is incorrectly classified as a sombrero. However, the network is only 50.91% certain of this classification. When looking at the visualization input image, the shape of the watermelon plate does resemble a sombrero, and the saliency map highlights an area that is consistent with such a shape. We also see that the head of the woman is highlighted, and signifying that the network has learned that faces are prominently featured in images of sombreros.

6.1.4 Deconvolutional Network

For the MNIST network, four simple visualizations produced using a deconvolutional network were presented. The visualizations are shown in Figure 5.9, and clearly illustrate that the feature maps are looking for edges at various locations in the input image. As described in Section 5.1.1, the MNIST network is not particularly suitable to showcase the deconvolution technique, since the network is fairly small. The VGG network is significantly larger, and is therefore a better option for demonstrating the visualization technique.

Figure 5.15 presents visualizations from a deconvolutional network for feature maps in the last pooling layer of the VGG network. Naturally, these feature maps seem to be looking for complex features such as wheels (number 348 and 450) and faces (number 380). In Figure 5.16 we can confirm our assumptions for feature map number 348 and 380, as they focus on the wheels and face present in the other two visualization input images as well.

In Figure 5.17, we see visualizations using the deconvolutional technique for a lower layer. The visualizations shows that the feature maps of this layer search for more specific, but still complex, features. The comparison shows that the feature maps are looking for different parts of a wheel. Visualizations for the lowest pooling layer of the VGG network are seen in Figure 5.18. These feature maps are searching for basic features such as colors and edges. For instance, feature map number 18 obviously looks for the color pink, feature map number 29 looks for a thin line, and number 47 for a rounded corner, and so on.

6.1.5 Deep Visualization

Figure 5.10 shows the evolution of the deep visualizations for the various visualization input images of the MNIST network. The same trend is evident here as in the other techniques described: the biggest changes happen between the first stages. We also see that as the training progresses and the network sees more examples, the deep visualizations actually seem to look less like the actual number. For instance, digits 1 and 5 start looking slightly distorted at stage 10. This may be because the network is getting more generalized and learns that the digits can have large variations. The digit 1 might be written straight from top to bottom, or slightly tilted from right to left. Another detail worth noting is that the right line of digit 4 is much more noticeable on the upper half than on the bottom half. We saw this trend in the saliency maps as well, where the network deemed some regions of the digits more important than others. It is evident that the bottom of the line is not a critical factor for classifying a digit as 4, likely because it is present in other digits as well, such as 7 and 9.

Deep visualizations of units in a lower layer of the MNIST network are shown in Figure 5.11. The visualization of the units of the lower layers indicate how the characteristics found in the output unit visualizations are constructed. We can see patterns in these visualizations that resemble parts of the visualizations in the output layer, but none of them look like complete digits. For instance, unit 24 resembles the stage 10 visualizations of digit 7. Unit 0 looks like digit 0 with its rounded parts in the bottom left and top right, but it also shows a vertical line at the top similar to the vertical line of digit 7. Unit 40 shows four white dots at the top that many of the output visualizations seem to contain at least one of, e.g. digits 2, 3, 4 and 6. The black unit 56 is most likely the result of a dead filter. Figure 5.12 displays deep visualizations of units in an even lower layer. Here, unit 0 resembles the previously described unit 40, and unit 96 looks somewhat like the digit 3.

The deep visualizations of selected units in the output layer of the VGG network, seen in Figure 5.19, are significantly more complicated. Most of the output classes are somewhat straightforward to identify. For instance, the tarantula class visualization contains furry, insect-like legs and prominent spider shapes, and the starfish class visualization shows star-shaped objects with a scale-like texture. In the bow tie class visualization, there are shapes clearly resembling bow ties. In addition, various patterns such as stripes, dots and plaid can be identified. This illustrates the invariance of the class, in that a bow can have many different patterns. Some of the visualizations, however, contain parts that we may not immediately see as directly related, but that are common in the context of the class.

A good example is the volleyball class visualization. There are several round shapes in the image, but they are not particularly dominant. We can also see what resembles hands, stretching up from the bottom. In addition, the visualization is covered in a texture that appears to be a net. This shows that it is the combination of several objects that results in a classification. The presence of other objects related to a specific output class improves the certainty that the image belongs to that class. This can also be seen in both the bubble and umbrella class visualizations. Parts of faces are noticeable, because there is usually a person blowing the bubbles, or a person standing beneath the umbrella. The umbrella class visualization also contains the texture of raindrops, which makes sense for umbrellas.

The deep visualization technique incorporates an element of randomness into the creation of visualizations. This is shown in Figure 5.20. The tarantula class visualization is produced twice with the exact same hyperparameter settings. We see that the images are not identical, but they definitely illustrate similar characteristics. Each of the images represent what the chosen unit is looking for, and the variance in the produced images represent the network's invariance. In Figure 5.21, four versions of the tarantula class version is produced, this time with various combinations of hyperparameters. These show how the hyperparameters can be altered to emphasize different characteristics, like the high frequency information in the top-right image or the most crucial patterns in the bottom-right image. The top-left images display a small set of important regions, while the bottom-left reveals typical low frequency patterns.

Figure 5.22 shows visualizations of the second fully connected layer. These are more challenging to interpret, since we do not have a definite answer of what the unit is looking for, like we did in the output layer. Also, the extent of the output classes makes it difficult to identify similarities to the output class visualizations. By studying the visualizations closely, we can still speculate on what the units are looking for. The first two visualizations seem to resemble birds of some kind, which is reasonable knowing that there are a number of different bird species among the output classes. The third visualization contains something that resembles leaves or insect wings.

In Figure 5.23, we can see the visualizations of the first fully connected layer. These appear to be more intense and less specialized than the second fully connected layer, which can be attributed to the fact that it is directly connected to every unit in the last convolutional layer. Consequently, it has the potential to detect a wide variety of features at once. In unit 2048, we see the resemblance of furry bodies, while unit 3072 show parts of snake heads, particularly at the top left corner. Recall that the features of the units in underlying layers are further combined in order to construct the final features of the output layer. The lower layer units would then contain features that may appear in several different output classes.

Figure 5.24 illustrate how the deep visualizations change in size and complexity as they go further down into the network. Units in block 2 clearly identifies simpler features than block 5. In Figure 5.25, we see how changing the learning rate and number of iterations affect the units of a lower layer. Typically, a higher learning rate and more iterations lead to more pronounced features in the visualizations, but at a certain point they are oversaturated, as seen in the leftmost image in 5.25. Finally, Figure 5.26 once again confirms that the lower layers look for the more fundamental features. For instance, unit (56, 56, 8) looks for pixels with a prominent value compared to its surroundings, and unit (56, 56, 48) detects edges.

6.1.6 Usefulness of the Visualization Tool

The visualization tool incorporates the presented visualization techniques in a simple web interface, as seen in the user manual in Appendix B. It allows users to browse the various visualizations while the network is training. By interpreting the results of Section 5.1, we have seen that there is undoubtedly insight to be gained from using these visualization techniques. For a fully trained network, the visualizations provide indications on how the ANN is behaving and why it behaves as it does. We have also seen that examining the visualizations during training can deepen the understanding of how the network progresses and when the learning process seem to be at an end, marked by diminishing changes to the visualizations. The training progress provides a straightforward indication on how well the network is performing, while the other techniques focus on uncovering what the various units in the network are actually looking for or find important, using different perspectives. The knowledge acquired from studying each of the visualization techniques may overlap, but by combining the various approaches, we can gain a greater overall intuition.

We chose to showcase the visualization techniques using well-known example ANNs, which are already known to achieve good results, to concentrate our focus on what the visualizations reveal about their behaviour. The visualizations provide insights into the networks, whether they perform well or poorly. Understanding how a network behaves, and why, makes it possible to identify shortcomings of less successful networks, and further to address these by making informed decisions on improvements.

Many of the resulting visualizations were presented side-by-side, as comparison can be useful and lead to even further insight. Although looking at visualizations for a number of visualization input images is helpful, the tool currently only allows a single visualization input image for each network. This feature was given a lower priority compared to supplying a wide range of visualization techniques and making them easy to use.

6.2 Case Study in Face Recognition

The case study networks included in Section 5.2 do not all represent the most promising architectures. We have three architecture types: the baseline, the architecture with extra input, and the architecture with extra output. For each of these, three configurations were included: a minimal, a preferred, and an interesting alternative. The minimal configurations were A, D and G, while the preferred were B, E and H, and lastly, the alternate ones were C, F, and I. Before comparing the three architecture approaches, we first compare the chosen configurations within each approach.

6.2.1 Baseline Networks

The baseline networks were created using configurations A, B and C. The difference between these networks is their depth, where B is deeper than A, and C is deeper than B. As can be seen in Table 5.1, the accuracy achieved by the minimal network A is relatively high. This, combined with the fact that this network only has a single weight layer, demonstrates the network's ability to easily discern valuable information from the features outputted from the pretrained network. By extension, it confirms the pretrained network's suitability as a feature extractor. To improve the performance, B adds depth, resulting in a more complex classifier. The increased complexity allows the network to interpret the feature input more thoroughly and better tune itself towards images in the IMFDB dataset. The result is a rise in accuracy and a decline in loss compared to A. C, however, which is the deepest baseline network, achieve poorer results than B in general, and has larger loss values than A. A reasonable explanation is that the added parameters enable the network to overfit to the small dataset. The improved accuracy over A is likely caused by outlier examples being correctly classified with low confidence.

6.2.2 Extra Input Networks

Configurations D, E and F were used to create the extra input networks. These differ not only in depth, but also in where the additional input is supplied. E and F have the equally deep, with D being shallower than both. Using a minimal configuration, D has no other option than to inject the extra input right at the start of the network. E and F introduce the expression input at different depths, which marks the main distinction between their architectures. From Table 5.1, we can see that the complexity gained by adding more layers has a beneficial effect on performance, as both the deeper networks outperform the shallow one, in all aspects. The network from configuration D has inferior performance due to its sole weight layer, the output, which needs to make sense of the combined inputs without any additional processing. E and F does have extra processing, but distinct kinds of processing. In F, the expression input is provided early on in the network, forcing every weighted layer in the model to process the information from the feature and expression inputs simultaneously. E, however, injects the expression input later, leaving earlier layers free to exclusively process the feature input. With the performance of E being unequivocally better than F's, the processing of feature input in isolation appear to have advantageous effects. A possibility is that the computations on the feature input alone refines the features, which then eases the subsequent simultaneous processing.

6.2.3 Extra Output Networks

The extra output network were based on architecture configurations G, H and I. They have varying depth, and different specialized processing parts. H and I are equally deep, both having more layers than G. These two are also similar in the early parts of the networks, with two weighted layers that are used for classifying both identification and expression. However, H utilizes additional specialized processing for the expression output, while I uses the equivalent for the identification output. G has no such components, and rather computes both classification outputs immediately. The identification and expression specific results can be found in Table 5.2 and Table 5.3, respectively. Together, these results reveal how the architectural decisions affect performance.

H stands out as the top performer, with superior values for both loss and accuracy when classifying identification. Looking at the expressions metrics, it also has the lowest expression output loss. Despite this, its accuracy in expression classification is not far off from I's. As mentioned before, this could be caused by the larger loss network, in this case I, correctly classifying outlier examples with low confidence. In H, the feature input is passed through two common weight layers before being presented to the separate output layers. These layers are influenced by both output layers during training, and are subsequently forced to produce information that is valuable to the two classification tasks simultaneously. It is here the use of expression data has the possibility to influence the performance of the network, by potentially revealing a pattern in how the variations in facial features that map to the same identity coincide with the variations in the expected expression outcome. The specialized processing for expression classification in H comes from an additional weight layer which is only affected by the expression output layer when training. Network I has similar shared layers as H, but uses specialized processing for the identification classification instead of for expressions. The common layers in I do improve expression performance compared to G, but, peculiarly, these layers combined with the specialized processing do not contribute to an definite advantage in identification classification. Again, I shows somewhat similar accuracy, but has a larger loss. As mentioned in Section 4.2.6, the lack of an identification performance boost is likely caused by overfitting due to the specialized processing.

6.2.4 Comparing Approaches

To compare the extra output approach to the others, we focus on the identification performance. When looking at the architectures of the networks, D and G resemble the baseline configuration A. Of these, D is the most successful performance-wise, with A and G showing similar performances. The latter comes as no surprise, considering that the input in G is not processed by any weight layers that are shared among the identification and expression output layers. Consequently, there are no common layers which both affect in the learning procedure, and the classification of identification and expression is executed strictly separately. Classifying the identification is then performed similarly in A and G. The sole difference between A and D is the extra input D receives. D's utilization of this additional information is the only architectural explanation for its increase in performance.

The situation is similar to how E and F compare to B. The enhanced performance of E and F over D solidifies the interpretation of B's improvement over A, that the added layers allow for increased complexity and potential tuning to our particular input. However, in the extra input architectures, the more complex systems have access to more information, and, as implied by the advance in performance, they find a meaningful use for it. With the addition of the expression information being the only architectural difference from B, it must be the cause of the performance gain. All of the baseline configurations can then be improved upon by including an additional expression input.

The last baseline configuration, C, is comparable to H and I. Among these, H has the most promising results, with C performing slightly above I, only notably surpassing it when regarding the loss calculated from the validation set. While H is as deep as C, its full depth is only utilized to classify the expression, and the identification result is computed after the first two weight layers. Since we are considering the extra output architectures based their identification accuracy, H could be compared to B, by disregarding the expression specific processing part. These two have quite similar results, with H having slightly higher accuracy, but also slightly larger loss values. Because of the meager differences, we cannot decisively say that one configuration is preferred over the other based on their performance. However, if we include complexity, B is the better choice. Not only does its architecture contain less parameters, it also requires less data in training.

Of the three approaches, the extra input architectures have the most impressive results. Every configuration in this approach outperforms their corresponding baseline alternatives. Compared to the extra output architectures, they are also generally superior, with the exception of the minimal configuration D versus H. However, the most interesting architectures consist of the preferred configuration of each approach, namely B, E and H. E is definitively the best choice among these, with overall better performance measurements, proving that the extra input architecture is the approach to prefer. In addition, B is favored over H, as previously discussed, making the extra output architecture the least promising alternative, even after the vanilla baseline approach. These results demonstrate that a face recognition system can be improved by exploiting facial expression data, and that an advantageous approach is to include the expression information as an additional network input.

6.2.5 Experienced Limitations

It is important to note that the performance gain of the extra input architectures was moderate, as was the performance differences in general. An explanation for this is the humble size of the various architectures and their similarity to each other. While these qualities make the current performance differences more significant, it also raises the question about how the experimental approaches would alter the behaviour of larger networks, if they would perform similarly or if the differences would grow or diminish. In our case study, the size of the dataset prevents us from examining this interesting question, as attempts at creating a larger architecture consistently resulted in overfitted networks, despite heavy regularization. Additionally, without a substantially larger dataset, we did not have the ability to train a sophisticated feature extractor from scratch. Rather, we were confined to using a pretrained model. The model chosen did not have any training on expression data, because of a distinct lack of such alternatives.

With a more extensive dataset, a larger network could be trained with which the experimental approaches could be explored further. We would for instance have more options on where to input the expression data for the extra input architectures, which has shown to have an impact on performance in the smaller networks. Similarly, we could expand the amount of specialized processing for each output when using architectures with an additional expression output. The extra output networks could also benefit from using a feature extractor that had expression data included in its training.

Chapter 7

Conclusion and Future Work

In this chapter we offer a conclusion based on the results in Chapter 5 and the discussion in Chapter 6, both for the implemented visualization tool and the case study. Additionally, we include pointers to interesting directions for future research.

7.1 Conclusion

The goal of the implemented visualization tool was to help researchers improve their understanding of ANNs. Several visualization techniques developed for monitoring performance and visualize behaviour were examined: training progress plots, layer activations, saliency maps, deconvolutional networks and deep visualization. By exploring existing tools for visualizing and facilitating the training process of an ANN, we were able to identify shortcomings in the selection. Our findings showed that although the complete selection provided valuable functionality, there existed a gap between them that we wanted to bridge. Specifically, none of the tools combined network management with advanced real-time visualizations.

Consequently, we developed a visualization tool that addressed these issues[1]. We chose to implement our tool using the Keras library, which allowed us to support both TensorFlow and Theano. The tool provides a simple web interface where users can upload and execute Keras network scripts, visualizing data produced during training with the aforementioned visualization techniques. Taking advantage of Keras' custom callbacks, we created an API that allows users to easily incorporate the visualization techniques into their scripts. The visualization tool is implemented in loosely coupled modules, enabling modifiability. By testing the visualization tool on two example networks and interpreting the visualizations produced, we have demonstrated the tool's ability to facilitate a deeper understanding of ANNs and their behaviour.

[1]Found at https://github.com/mikaelbj/training-visualizer

For the case study, we were interested in how a face recognition system could be improved by utilizing facial expression data. To that end, we wanted to explore how face recognition networks behaved when expression data exploitation was incorporated into their architectures. We constructed experimental networks based on three architectural approaches: a baseline with no utilization of expression information, an architecture with an additional expression input, and an architecture with an additional expression output. Through incremental improvements of the experimental network architectures, we discovered nine interesting configurations, three of which represented the preferred architecture for each approach. Comparing the loss and accuracy of the nine allowed us to better understand how various architectural decisions affected their performance, as well as identify an approach that improved the face recognition system.

The performance measures revealed that the extra input architectures were more successful in classifying identification than their baseline counterparts. The chosen configurations had a sole architectural distinction compared the the baseline architectures, namely the additional input. This implies that the enhanced performance came as a direct result of the inclusion of the expression data. Consequently, the case study has shown that employing facial expression data as an additional network input has the potential to improve a face recognition system. The extra output architectures had less favorable results, performing mostly worse than the configurations from the other approaches. It would be beneficial to explore these results further, examining the effects of the experimental approaches with a larger dataset, with emphasis on the promising extra input architectures.

To summarize, we have successfully implemented a tool that facilitates insight into ANNs using visualization, and completed a face recognition case study that provided interesting findings. We hope that these results can help further the research into ANNs applied on images, and face recognition systems.

7.2 Future Work

This section provides some pointers to future work based on our findings for both the visualization tool and the case study.

7.2.1 Visualization Tool

The visualization techniques described in this thesis does not represent an exhaustive collection. Adding even more visualization options could further improve the understanding of ANNs. Some examples are the visualization of weights, plots of learning rate, and an improved version of saliency maps using guided backpropagation[2]. The tool could also be extended to work for a broader set of networks, for instance networks with several inputs.

Examining the resulting visualizations demonstrated that additional insight can be uncovered by comparing visualizations at different stages, or even of different visualization input

[2]https://github.com/Lasagne/Recipes/blob/master/examples/Saliency%20Maps%20and%20Guided%20Backpropagation.ipynb

images. A useful extension could thus be to incorporate the possibility of uploading several images at execution. The presentation of the visualizations could also be significantly improved to allow for better comparison. Another valuable feature that could be added is the option of visualizing a prediction using a fully trained network. The functionality for producing the visualizations is already implemented, but needs to be incorporated in a different way than through callbacks if they are to be used in prediction.

Performance was not considered a serious concern while implementing the visualization tool. Some of the visualization techniques are computationally costly and will therefore take some time to perform, regardless of the implementation. However, in cases where there are a large number of visualizations to be presented, the presentation should be done as smooth and quick as possible. In this regard, Bokeh might not be the best visualization library, due to an experienced low efficiency, despite providing other useful qualities. Since the tool was created with modifiability in mind, replacing Bokeh with another visualization library would be fairly uncomplicated.

7.2.2 Case Study in Face Recognition

The case study raised an interesting question concerning how the assessed experimental approaches would affect performance when applied to larger networks. This presents a compelling idea for future study, but requires a dataset of considerable size to investigate. It would be natural to focus further efforts on the approach that utilizes expression data as input. With an appropriately extensive dataset, it would also be interesting to train full networks, which provide more possibilities than the current separation of a network into a pretrained feature extractor and classifier.

Additionally, the effect of the expression range could be researched, as the size of the expression domain may influence how the networks use the expression data. To alter the domain size, expressions could be merged, for example by considering sadness and anger as a single expression. A wider array of expressions may allow the networks to better capture the fine feature variations and thereby offer greater precision, but it may also make them more difficult to learn. In contrast, a narrower range could present an easier learning task, but the coarser expression representations may restrict the performance increase by disregarding the subtle differences between the merged expressions.

Bibliography

[1] L. Fei-Fei, A. Karpathy, and J. Johnsen. Convolutional Neural Networks for Visual Recognition - Neural Networks Part 1: Setting up the Architecture. `http://cs231n.github.io/neural-networks-1/`. Accessed: 30.03.2017.

[2] D. D. Zhang. *Biometric Solutions: For Authentication in an E-World*. 2002.

[3] T. Huang, Z. Xiong, and Z. Zhang. *Handbook of Face Recognition*. 2011.

[4] D. N. Parmar and B. B. Mehta. Face Recognition Methods & Applications.

[5] B. J. Oates. *Researching Information Systems*. SAGE Publications, 2006.

[6] S. March and G. Smith. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4):251–266, 1995.

[7] V. Vaishnavi and B. Kuechler. Design Science Research in Information Systems.

[8] L. Fei-Fei, A. Karpathy, and J. Johnsen. Convolutional Neural Networks for Visual Recognition - Commonly Used Activation Functions. `http://cs231n.github.io/neural-networks-1/#actfun`. Accessed: 12.05.2017.

[9] G. E. Hinton V. Nair. Rectified Linear Units Improve Restricted Boltzmann Machines. 2010.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015.

[11] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout Networks. 2013.

[12] C. M. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[13] N. Srivastava, G. Hinto, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 2014.

[14] L. Fei-Fei, A. Karpathy, and J. Johnsen. Convolutional Neural Networks for Visual Recognition - Softmax. `http://cs231n.github.io/linear-classify/#softmax`. Accessed: 19.06.2017.

[15] S. Ruder. An Overview of Gradient Descent Optimization Algorithms. 2016.

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. 2016.

[17] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. 2016.

[18] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson. Understanding neural networks through deep visualization. 2015.

[19] L. Fei-Fei, A. Karpathy, and J. Johnsen. Convolutional Neural Networks for Visual Recognition - Understanding and Visualizing Convolutional Neural Networks. `http://cs231n.github.io/understanding-cnn/`. Accessed: 20.10.2016.

[20] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. 2014.

[21] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks.

[22] M. D. Zeiler, G. W. Taylor, and R. Fergus. Adaptive Deconvolutional Networks for Mid and High Level Feature Learning.

[23] L. Fei-Fei, A. Karpathy, and J. Johnsen. Convolutional Neural Networks for Visual Recognition - Regularization. `http://cs231n.github.io/neural-networks-2/#reg`. Accessed: 12.05.2017.

[24] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks?

[25] L. Fei-Fei, A. Karpathy, and J. Johnsen. CS231n Convolutional Neural Networks for Visual Recognition: Transfer Learning. `http://cs231n.github.io/transfer-learning/`. Accessed: 20.05.2016.

[26] P. Ekman and W.V. Friesen. Facial Action Coding System. 1978.

[27] T. Kanade, J. F. Cohn, and Y. Tian. Comprehensive database for facial expression analysis. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=840611`.

[28] P. Ekman. Universal Facial Expressions of Emotions.

[29] T. Huang, Z. Xiong, and Z. Zhang. *Handbook of Face Recognition*. 2011.

[30] M. Liu, S. Li, S Shan, and X. Chen. AU-inspired Deep Networks for Facial Expression Feature Learning.

[31] P. Burkert, F. Trier, M. Z. Afzal, A. Dengel, and M. Liwicki. DeXpression: Deep Convolutional Neural Network for Expression Recognition.

[32] P. Lucey, J. F. Cohn, T. Kanade, J. Saragih, Z. Ambadar, and I. Matthews. The Extended Cohn-Kanade Dataset (CK+): A complete dataset for action unit and emotion-specified expression. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5543262`.

[33] A. M. Bronstein, M. M. Bronstein, and R. Kimmel. Expression-invariant representations of faces.

[34] X. Li, G. Mori, and H. Zhang. Expression-Invariant Face Recognition with Expression Classification.

[35] H. Mliki, E. Fendri, and M. Hammami. Face Recognition Through Different Facial Expressions.

[36] F. Schroff, D. Kalenichenko, and J. Philbin. FaceNet: A Unified Embedding for Face Recognition and Clustering.

[37] Y. Taigman, M. Yang, M. A. Ranzato, and L. Wolf. DeepFace: Closing the Gap to Human-Level Performance in Face Verification.

[38] Y. Sun, D. Liang, X. Wang, and X. Tang. DeepID3: Face Recognition with Very Deep Neural Networks.

[39] Y. Sun, X. Wang, and X. Tang. Deeply learned face representations are sparse, selective, and robust.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks.

[41] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions.

[42] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition.

[43] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[44] C. Lu and X. Tang. Surpassing Human-Level Face Verification Performance on LFW with GaussianFace.

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, and J. Shlens. Rethinking the Inception Architecture for Computer Vision.

[47] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks For Large-Scale Image Recognition.

[48] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep Face Recognition. 2015.

[49] L. Wolf, T. Hassner, and I. Maoz. Face Recognition in Unconstrained Videos with Matched Background Similarity. https://www.cs.tau.ac.il/~wolf/ytfaces/WolfHassnerMaoz_CVPR11.pdf.

[50] S. Setty, M. Husain, P. Beham, J. Gudavalli, M. Kandasamy, R. Vaddi, V. Hemadri, J. C. Karure, R. Raju, Rajan, V. Kumar, and C. V. Jawahar. Indian Movie Face Database: A Benchmark for Face Recognition Under Wide Variations. Dec 2013.

[51] F. Chollet. Keras documentation. https://keras.io/, 2015.

[52] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks.

Appendix A

Installation & Setup

This explains how to install and set up the visualization tool ready for use. We assume that the user has Python 3.5 installed.

A.1 Installation

The first thing you need to do is download or clone the GitHub repository to your computer. Open a command line interface and navigate to the main directory of the repository, `training-visualizer`.

Install all required packages by running the following command in the command line interface:

```
pip install -r visualizer/requirements.txt
```

Note: If you already have an installation of the Python Imaging Library (PIL) package, you will get a failure when trying to install the Pillow package listed in the requirements file. Remove Pillow from the requirements file and rerun the command above.

A.2 Configuration and Setup

The visualizer folder of the project contains a configuration file that can be used to configure some settings for the application, most of which are not of importance for a typical user. However, if you are using a different command than `python3` to run the desired Python executable, it is crucial that you change the `PYTHON` variable to the actual command you are using.

After the application has been properly configured, we can start performing the necessary setup. To make this process easier for the user, we have created bash scripts for Linux and MacOS, and batch scripts for Windows, containing the necessary commands.

A.2.1 Linux and MacOS

For Linux and MacOS, the only thing we need to set up the visualization tool is initializing the database. The environment setup is included in the scripts for starting the application, which we will see in a later section. The command for initializing the database is as follows:

```
source linux_macos/init_db.sh
```

A.2.2 Windows

For Windows, we need to both set up the environment and initialize the database. This is done by running these two batch files in the command line:

```
windows/init_env.bat
windows/init_db.bat
```

A.3 Starting the Visualization Tool

The application consists of two separate processes in order to function correctly: a Flask application server, and a Bokeh visualization server. We will now demonstrate how to start these servers.

A.3.1 Linux and MacOS

In the same command line window that you executed the database setup command, run the following in order to start the Flask application server:

```
source linux_macos/start_flask.sh
```

Open a new command line window, navigate to the same folder, and then run this command to start the Bokeh visualization server:

```
source linux_macos/start_bokeh.sh
```

A.3.2 Windows

In the same command line window that you executed the environment and database setup commands, run the following batch file in order to start both the Flask application server and the Bokeh visualization server:

```
windows/start_servers.bat
```

Navigate to `localhost:5000` in a web browser (we recommend Google Chrome), and the visualization tool login page should be displayed. For more information on how to proceed, see appendix B.

Appendix B

User Manual

Note: This appendix explains how to use the web interface of the visualization tool. Remember that certain parts of the functionality requires that you have added the available callbacks to your Keras network script. We refer the user to appendix C for the callbacks API, as well as an explanation of how to use them in a script.

The web interface is fairly simple to use. At all times, the top of the page will show a navigation bar with links to the available pages. When clicked, these links take you to the corresponding page. The interface will display error feedback for any illegal action you might take, such as trying to log in with a mismatching username and password, or trying to upload a file with a filename that is not unique. Similarly, feedback for successful activity, like uploading a file, is also shown.

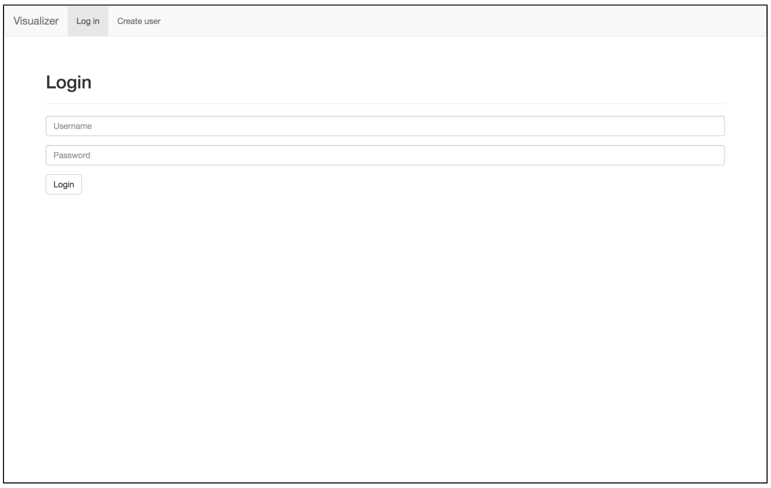
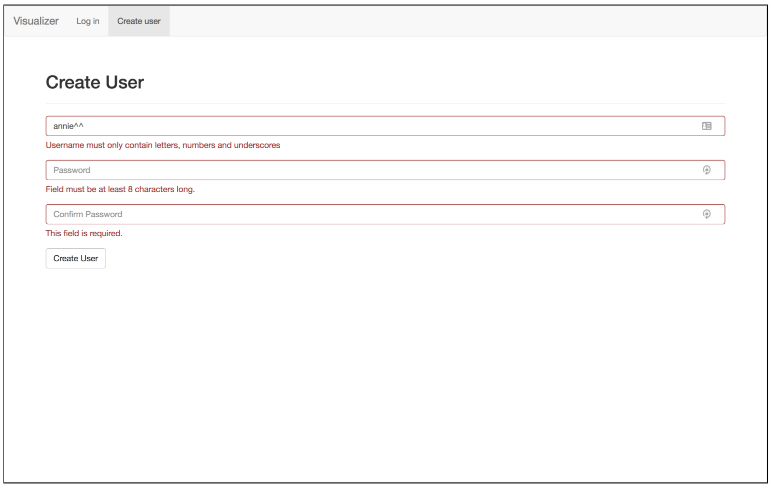


Figure B.1: Login



Create User

annie^^

Username must only contain letters, numbers and underscores

Password

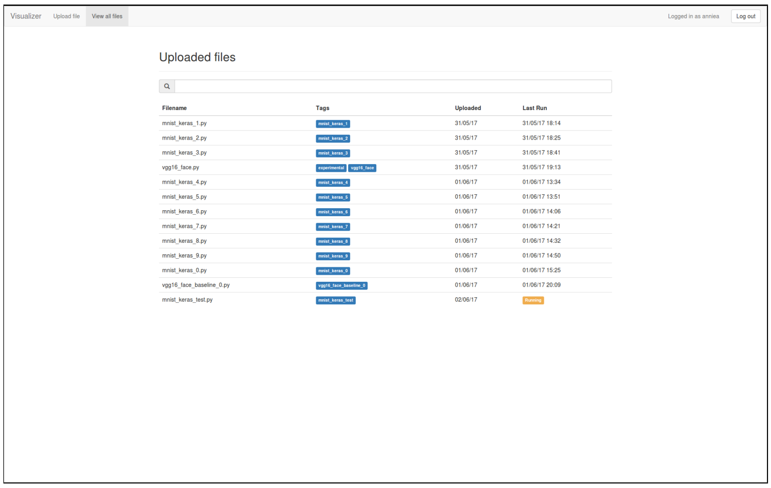
Field must be at least 8 characters long.

Confirm Password

This field is required.

Figure B.2: Creating a new user

When starting the visualization tool, you will be shown a login page as in **Fig. B.1**. If you do not already have a user, you can click the ”Create user” link to create a new user for the application. Usernames must be between 5 and 20 characters long, and can only contain alphanumerical characters and underscores. The username must also be unique. The password must be minimum 8 characters and can only contain alphanumeric characters. As seen in **Fig. B.2**, the interface will provide you with descriptive feedback if any of these restrictions are broken. When you have successfully created a user, you are automatically redirected back to the login page, where you can log in to start using the visualization tool.

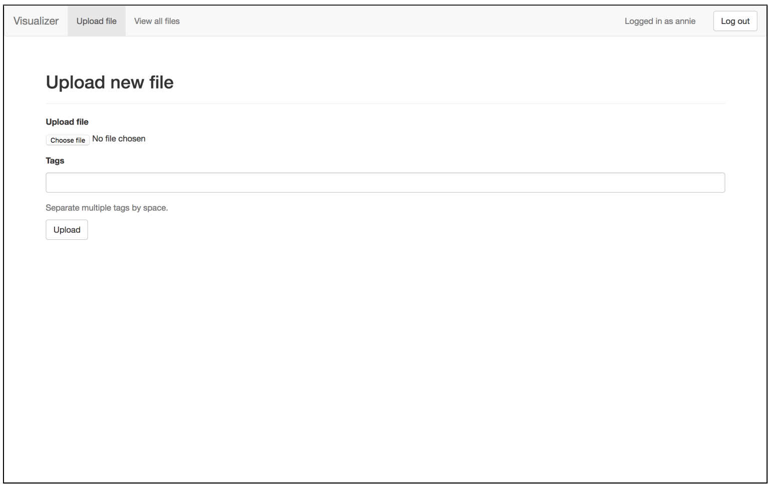


Uploaded files

Filename	Tags	Uploaded	Last Run
mnist_keras_1.py	mnist_keras_1	31/05/17	31/05/17 18:14
mnist_keras_2.py	mnist_keras_2	31/05/17	31/05/17 18:25
mnist_keras_3.py	mnist_keras_3	31/05/17	31/05/17 18:41
vgg16_face.py	experimental vgg16_face	31/05/17	31/05/17 19:13
mnist_keras_4.py	mnist_keras_4	01/06/17	01/06/17 13:34
mnist_keras_5.py	mnist_keras_5	01/06/17	01/06/17 13:51
mnist_keras_6.py	mnist_keras_6	01/06/17	01/06/17 14:06
mnist_keras_7.py	mnist_keras_7	01/06/17	01/06/17 14:21
mnist_keras_8.py	mnist_keras_8	01/06/17	01/06/17 14:32
mnist_keras_9.py	mnist_keras_9	01/06/17	01/06/17 14:50
mnist_keras_0.py	mnist_keras_0	01/06/17	01/06/17 15:25
vgg16_face_baseline_0.py	vgg16_face_baseline_0	01/06/17	01/06/17 20:09
mnist_keras_test.py	mnist_keras_test	02/06/17	Running

Figure B.3: Uploaded files

When you have logged in, you will be directed to the page in **Fig. B.3** showing all your uploaded files. If there are no uploaded files yet, the page will suggest that you click on a link to upload a file. If you already have files uploaded, these will be displayed in a list with columns containing the filename, tags, upload date, and the date that the file was last ran. If the file is currently running, the last column will show a yellow label that reads ”Running”. You can sort the list by clicking on the various columns. A search bar is also provided, and it responds to filenames and tags. Clicking on an entry in the list will take you to the corresponding file’s overview page. First, we will describe the upload page.



Upload new file

Upload file

Choose file No file chosen

Tags

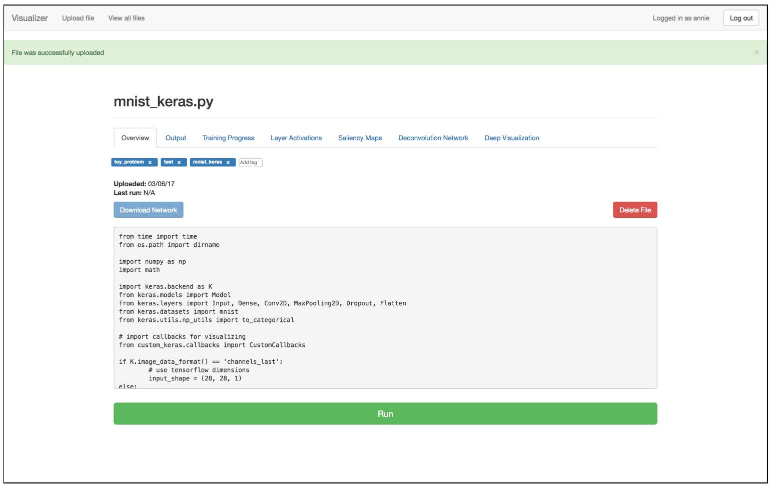
Separate multiple tags by space.

Upload

Figure B.4: Uploading a file

You can upload Keras network scripts from your computer on the file upload page, as seen in **Fig. B.4**. The file uploaded will retain its filename, which must be unique. To better organize your files, you can associate tags to the file before uploading. The tags are split at whitespaces, for instance ’neural network’ will result in two tags: ”neural” and ”network”. Use underscores if you want to add tags containing multiple words. The filename, without its extension, will be automatically added as a tag.

When a file is successfully uploaded, you are taken to the file’s overview page, shown in **Fig. B.5**. A submenu will be available with seven tabs: overview, console output, training progress, layer activations, saliency maps, deconvolutional network and deep visualization. The overview page shows some information about the file, as well as its contents. You have the option of downloading the generated network, assuming that the script includes the network saving callback and has been executed. You can also delete the file if you wish, along with all associated data. Lastly, you can run the file by clicking the green button that reads ”Run”. Upon clicking the run button, a popup window will appear and ask you to select an image to use for the visualizations. If an image has already been uploaded, that image will be available as a default, as seen in **Fig. B.6**. Upload a new one, or use the previous image and click ”Run”. Also note that if the file already has been



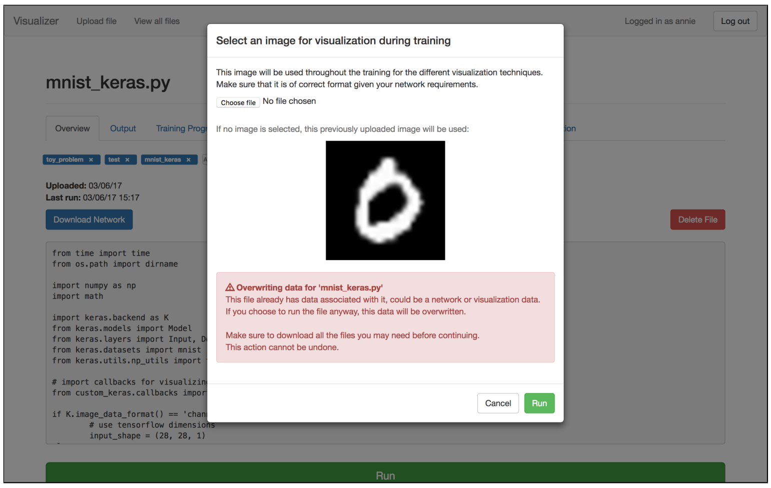
File was successfully uploaded

mnist_keras.py

Uploaded: 03/06/17
Last run: N/A

```
from time import time
from os.path import dirname

import numpy as np
import math

import keras.backend as K
from keras.models import Model
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout, Flatten
from keras.datasets import mnist
from keras.utils.np_utils import to_categorical

# import callbacks for visualizing
from custom_keras.callbacks import CustomCallbacks

if K.image_data_format() == 'channels_last':
        # use tensorflow dimensions
        input_shape = (28, 28, 1)
else:
```

Figure B.5: File overview



mnist_keras.py

Select an image for visualization during training

This image will be used throughout the training for the different visualization techniques.
Make sure that it is of correct format given your network requirements.

No file chosen

If no image is selected, this previously uploaded image will be used:

⚠ Overwriting data for 'mnist_keras.py'
This file already has data associated with it, could be a network or visualization data.
If you choose to run the file anyway, this data will be overwritten.

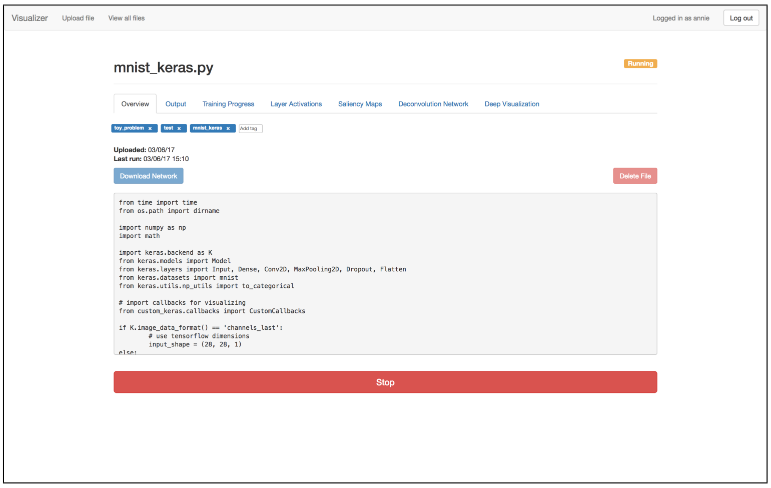
Make sure to download all the files you may need before continuing.
This action cannot be undone.

Uploaded: 03/06/17
Last run: 03/06/17 15:17

Figure B.6: Running a file

run, it will have data associated with it, for instance networks or visualization data files. Running the file again will delete all of the previously generated data, which the popup warns about. When the popup run button is clicked, the code in the file will be executed in a Python subprocess. When a file is running, it will have the yellow ”Running” label displayed at the top right of its page. You can also end the execution of the file at any time by clicking the stop button. A banner with a link to the file will be shown whenever one of your files completes its execution.

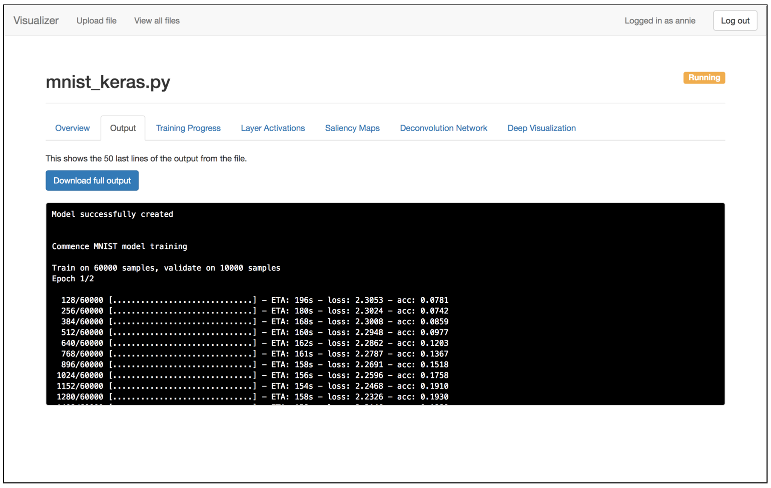
While a file is running, you can navigate to the various tabs to see different output and



Uploaded: 03/06/17
Last run: 03/06/17 15:10

```
from time import time
from os.path import dirname

import numpy as np
import math

import keras.backend as K
from keras.models import Model
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout, Flatten
from keras.datasets import mnist
from keras.utils.np_utils import to_categorical

# import callbacks for visualizing
from custom_keras.callbacks import CustomCallbacks

if K.image_data_format() == 'channels_last':
        # use tensorflow dimensions
        input_shape = (28, 28, 1)
else:
```

Figure B.7: File overview when file is running



This shows the 50 last lines of the output from the file.

```
Model successfully created


Commence MNIST model training

Train on 60000 samples, validate on 10000 samples
Epoch 1/2

  128/60000 [..............................] - ETA: 196s - loss: 2.3053 - acc: 0.0781
  256/60000 [..............................] - ETA: 180s - loss: 2.3024 - acc: 0.0742
  384/60000 [..............................] - ETA: 168s - loss: 2.3008 - acc: 0.0859
  512/60000 [..............................] - ETA: 160s - loss: 2.2948 - acc: 0.0977
  640/60000 [..............................] - ETA: 162s - loss: 2.2862 - acc: 0.1203
  768/60000 [..............................] - ETA: 161s - loss: 2.2787 - acc: 0.1367
  896/60000 [..............................] - ETA: 158s - loss: 2.2691 - acc: 0.1518
 1024/60000 [..............................] - ETA: 156s - loss: 2.2596 - acc: 0.1758
 1152/60000 [..............................] - ETA: 154s - loss: 2.2468 - acc: 0.1910
 1280/60000 [..............................] - ETA: 158s - loss: 2.2326 - acc: 0.1930
```

Figure B.8: Output of a running file

visualizations. The output tab in **Fig. B.8** shows the last 50 lines of the command line output of the file in real-time. The output is useful for viewing estimated time to training completion and epoch accuracy, as well as debugging unexpected halts in the execution. You can download the full output by clicking the button.

Note: The next tabs described require that you have added the corresponding callbacks to your code.

The training progress tab in **Fig. B.9** naturally shows the progress of the training. Two

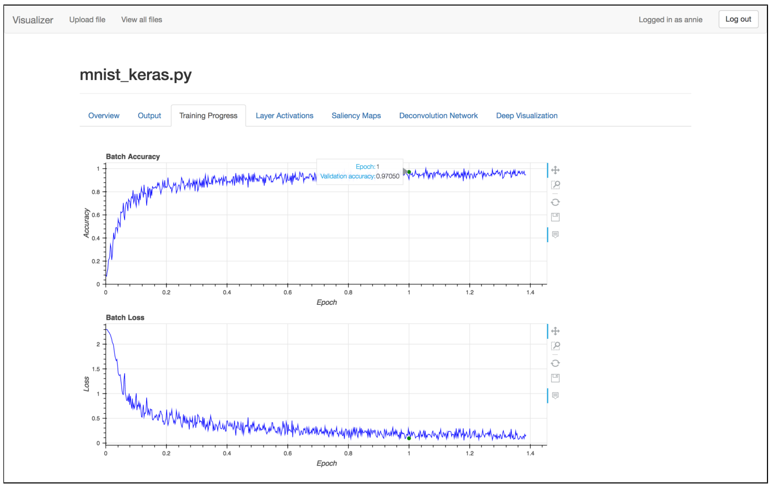


Figure B.9: Training progress

plots are available: one for the accuracy and one for the loss. The data shown is the accuracy and loss for each batch in the training (unlike the command line output which shows the average accuracy and loss for the current epoch) over the epoch share. This will give you an overview of how the accuracy and loss are changing through the training process. If you have enabled validation in your Keras code, the validation accuracy and validation loss will be displayed on the graph at each full epoch as a green, circular point. You can hover over these points to see the exact value if you have the hover tool selected on the toolbar. Other tools available are zooming and panning the plot, as well as saving the plot and resetting to the default view.

The next tab, seen in **Fig. B.10** and **Fig. B.11**, shows the layer activations. Note that this tab may take some time to load, depending on the size of your input image and the number of layers in your network. Once loaded, you can see the input image, followed by the activations for the different layers. Input, dropout and flatten layers are automatically excluded if not specified otherwise in the callback. For layers with 3D output, like convolutional and pooling layers, the activations of the filters are concatenated into grids separated by white borders. Layers with 1D output are also displayed in image format, with each output unit as a pixel. The activations can be used to determine what features a network learns at its various layers. Additionally, if some feature maps are entirely dark for many different visualization input images, so-called dead filters, this can be a warning that the learning rate is too high.

We can view the saliency map of the uploaded visualization image in the next tab, as shown in **Fig. B.12**. The left figure shows the original image, while the right figure shows the computed saliency map. The tools are linked so that if you zoom in on a region of the original image, the same zoomed in region will be displayed in the saliency map, and vice versa. The saliency map indicates to what degree each pixel in the visualization image

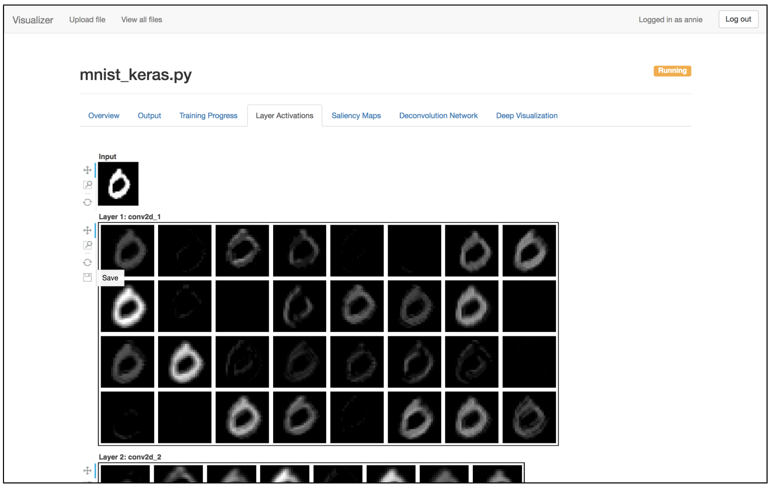


Figure B.10: Layer activations part 1

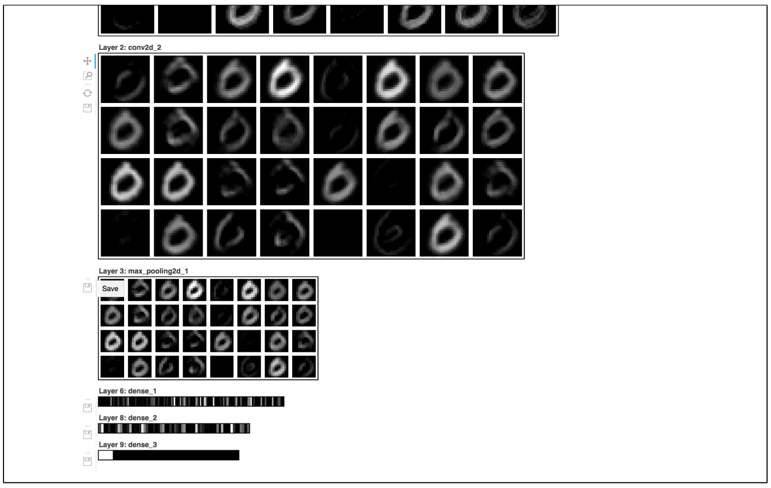


Figure B.11: Layer activations part 2

influences the specific class outcome. The important regions are easily identified by their brightness. The saliency map can help you to see which parts of the visualization image that your network deems more important when deciding upon the classification score chosen.

The next tab shows the deconvolutional network visualization, seen in **Fig. B.13**. The top figure shows the uploaded visualization image. Below the original image, there is a grid of figures showing the reconstructed input based on the feature maps selected for visualization. The visualizations show the pattern in the visualization image that was responsible

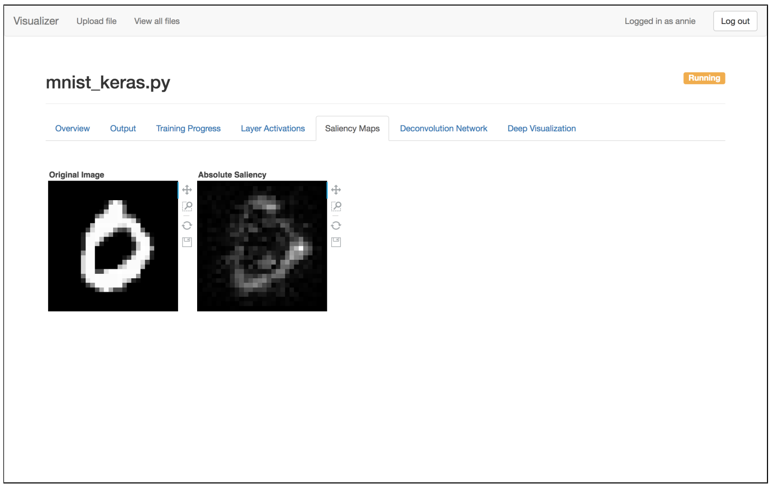


Figure B.12: Saliency map

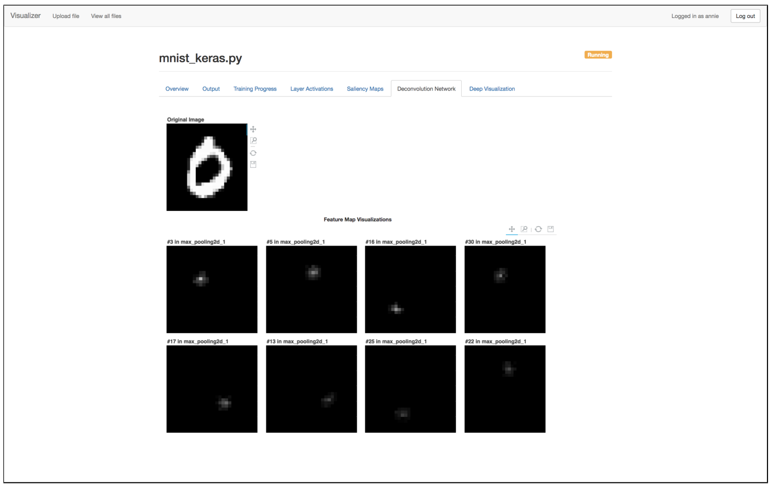


Figure B.13: Deconvolutional network

for eliciting activations in the specific feature map. This pattern can reveal what a certain part of the network finds important and what it is looking for in an image. The tools of the figures are linked together similarly to the saliency map figures. Additionally, if you click the save tool, all of the deconvolutional visualizations will be downloaded.

The final tab, as seen in **Fig. B.14**, shows the images created using deep visualization. This technique does not utilize the uploaded visualization image. The page presents a grid of figures, with a synthesized image for each network unit chosen for visualization. The synthesized images are optimized to maximally activate their corresponding unit. In other

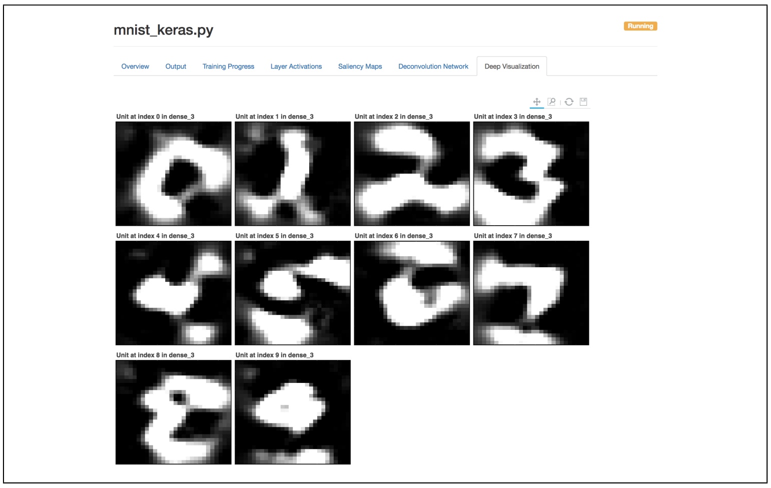

mnist_keras.py
Running
Overview
Output
Training Progress
Layer Activations
Saliency Maps
Deconvolution Network
Deep Visualization
Unit at index 0 in dense_3
Unit at index 1 in dense_3
Unit at index 2 in dense_3
Unit at index 3 in dense_3
Unit at index 4 in dense_3
Unit at index 5 in dense_3
Unit at index 6 in dense_3
Unit at index 7 in dense_3
Unit at index 8 in dense_3
Unit at index 9 in dense_3

Figure B.14: Deep visualization

words, they show what the selected units are looking for in an image.

For a more thorough explanation of each visualization technique, visit Section 2.3.

Appendix C

Callbacks API

This appendix will explain how to use the custom callbacks in your Keras network script, and provide you with an API. For a presentation of the various visualization techniques explaining the underlying theory, see Section 2.3 of the thesis.

C.1 Usage of Callbacks

The API includes a wrapper class that allows for adding and initializing the custom callbacks in a simple and straightforward manner. The first thing you need to do is to import the wrapper class:

```
from custom_keras.callbacks import CustomCallbacks
```

Then, you instantiate the wrapper class like this:

```
callbacks = CustomCallbacks(os.path.dirname(__file__))
```

Note that to be able to use the `dirname` method you need to have the `os` package imported. The argument of the wrapper class should always be as shown above and is needed for the application to save the visualization results in the correct location.

The wrapper class provides you with methods for adding each of the available callbacks. For instance:

```
callbacks.register_deconvolutional_network(3, 16, interval=10)
```

This shows the deconvolutional network visualization technique being added. For an explanation of the arguments, we refer you to the corresponding callbacks's description below.

When you are finished adding the callbacks that you wish to include, you can pass them to Keras' `.fit()` and `.fit_generator()` methods as follows:

```
model.fit(x, y, callbacks=callbacks.get_list())
```

You can also instantiate the callbacks separately, instead of using the wrapper class.

C.1.1 CustomCallbacks

```
custom_keras.callbacks.CustomCallbacks(file_folder,
                                       custom_preprocess=None,
                                       custom_postprocess=None,
                                       base_interval=10)
```

A wrapper class for callbacks. The arguments are arguments that two or more of the callbacks have in common. They will be passed on to the applicable callbacks when adding them.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.
- **custom_preprocess:** A preprocess function that will be applied to the visualization input image before use.
- **custom_postprocess:** A postprocess function that will be applied to the visualization output image before save.
- **base_interval:** The base interval that the visualizations will be computed at, unless specified otherwise. Many of the callbacks have their own interval argument that can be set individually.

Methods

The arguments of the wrapper methods correspond to the arguments of the associated callbacks, excluding `file_folder`, `custom_preprocess` and `custom_postprocess`, which are set in the wrapper instantiation. Specifying the `interval` argument for a certain callback will override the `base_interval` argument.

- **get_list:** Returns a list of all registered callbacks, to be passed to the Keras `.fit()` and `.fit_generator()` methods.
- **register_network_saver:** Registers the `NetworkSaver` callback.
- **register_training_progress:** Registers the `TrainingProgress` calback.
- **register_layer_activations:** Registers the `LayerActivations` callback.
- **register_saliency_maps:** Registers the `SaliencyMaps` callback.

- **register_deconvolutional_network:** Registers the `DeconvolutionalNetwork` callback.
- **register_deep_visualization:** Registers the `DeepVisualization` callback.

C.2 Available Callbacks

C.2.1 NetworkSaver

```
custom_keras.callbacks.NetworkSaver(file_folder)
```

Saves the network after each epoch.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.

C.2.2 TrainingProgress

```
custom_keras.callbacks.TrainingProgress(file_folder)
```

Saves training accuracy and loss after each batch. If validation is enabled, the callback will also save the validation accuracy and loss after each epoch. This callback assumes that accuracy is enabled as a metric in your Keras model.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.

C.2.3 LayerActivations

```
custom_keras.callbacks.LayerActivations(file_folder,
                                        exclude_layers=EXCLUDE_LAYERS,
                                        custom_preprocess=None,
                                        interval=10)
```

Produces the layer activations for each layer of the network, except the excluded layers.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.
- **exclude_layers:** A tuple of Keras layers to exclude from visualization. The default value `EXCLUDE_LAYERS` is a tuple containing `keras.layers.InputLayer`, `keras.layers.Dropout` and `keras.layers.Flatten`

- **custom_preprocess:** A preprocess function that will be applied to the visualization input image before use.
- **interval:** The interval that the visualization will be computed at.

C.2.4 SaliencyMaps

```
custom_keras.callbacks.SaliencyMaps(file_folder,
                                    custom_preprocess=None,
                                    custom_postprocess=None,
                                    interval=10)
```

Produces a saliency map of the uploaded visualization image showing to what degree the pixels of the image influenced the result.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.
- **custom_preprocess:** A preprocess function that will be applied to the visualization input image before use.
- **custom_postprocess:** A postprocess function that will be applied to the visualization output image before save.
- **interval:** The interval that the visualization will be computed at.

C.2.5 DeconvolutionalNetwork

```
custom_keras.callbacks.DeconvolutionalNetwork(file_folder,
                                            feat_map_layer_no,
                                            feat_map_amount=None,
                                            feat_map_nos=None,
                                            custom_preprocess=None,
                                            custom_postprocess=None,
                                            custom_keras_model_info=None,
                                            interval=100)
```

Produces visualizations of feature maps in the specified layer using a deconvolutional network and the uploaded visualization image. For simple convolutional models, the deconvolutional model can be automatically generated. For more complex convolutional models, the deconvolutional model created must be created by hand and passed to the callback.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.
- **feat_map_layer_no:** The number of the layer whose feature maps should be visualized.
- **feat_map_amount:** The number of feature maps to visualize. The algorithm will choose the top *N* maximally activated feature maps. If the argument is set to -1, all feature maps will be visualized. Note that either this or the `feat_map_nos` argument always needs to be specified.
- **feat_map_nos:** A list of numbers denoting which specific feature maps to visualize. Note that either this or the `feat_map_amount` argument always needs to be specified.
- **custom_preprocess:** A preprocess function that will be applied to the visualization input image before use.
- **custom_postprocess:** A postprocess function that will be applied to the visualization output image before save.
- **custom_keras_model_info:** Used to provide a deconvolutional model. Should be a tuple containing, in respective order, a deconvolutional Keras model based on your original model, a dictionary mapping from original model layer numbers to the corresponding deconvolutional model layer numbers, and an update method for the deconvolutional model which returns the updated deconvolutional model and layer map. If no update is needed, input an empty method.
- **interval:** The interval that the visualization will be computed at.

C.2.6 DeepVisualization

```
custom_keras.callbacks.DeepVisualization(file_folder,
                                         units_to_visualize,
                                         learning_rate,
                                         no_of_iterations,
                                         l2_decay=0,
                                         blur_interval=0,
                                         blur_std=0,
                                         value_percentile=0,
                                         norm_percentile=0,
                                         contribution_percentile=0,
                                         abs_contribution_percentile=0,
                                         custom_postprocess=None,
                                         interval=1000)
```

Produces deep visualizations from the units selected. Involves a number of optional arguments for regularization values that can be set to employ various regularization techniques in order to produce interpretable visualizations. The regularization techniques available are L_2 decay, Gaussian blur and clipping based on specific attributes. Using the default value of a specific regularization technique will disable it. Some useful combinations of the regularization values can be seen in **Table 2.1**.

Arguments

- **file_folder:** Should always be `os.path.dirname(__file__)`.
- **units_to_visualize:** A list of tuples describing which units to be visualized. The tuples are on the form `(layer_no, unit_index)` where `unit_index` is a number for layers with 1D output, like `keras.layers.Dense`, or a 3D tuple for layers with 3D output, like `keras.layers.Conv2D`.
- **learning_rate:** The learning rate for updating the visualization image.
- **no_of_iterations:** The number of iterations to perform image optimization.
- **l2_decay:** The strength of the L_2 decay regularization technique. Prevents a small number of extreme pixel values from dominating the output image. Requires values in range [0.0, 1.0].
- **blur_interval:** The interval of employing the Gaussian blur regularization technique. Used to penalize high frequency information in the output image. Note that both this and the `blur_std` argument needs to be specified to enable Guassian blurring. Requires values in range [0, inf).
- **blur_std:** The standard deviation for the Gaussian blur kernel in the interval. Used to penalize high frequency information in the output image. Note that both this and the `blur_interval` argument needs to be specified to enable Guassian blurring. Requires values in range [0.0, inf).
- **value_percentile:** The value percentile limit. Used in clipping regularization to induce sparsity by setting pixels with a small absolute value to zero. Requires values in range [0, 100].
- **norm_percentile:** The norm percentile limit. Used in clipping regularization to induce sparsity by setting pixels with small norm to zero. Requires values in range [0, 100].
- **contribution_percentile:** [0, 100] The contribution percentile limit. Used in clipping regularization to induce sparsity by setting pixels with small contribution to zero. Requires values in range [0, 100].
- **abs_contribution_percentile:** The absolute contribution percentile limit. Used in clipping regularization to induce sparsity by setting pixels with small absolute contribution to zero. Requires values in range [0, 100].

- **custom_postprocess:** A postprocess function that will be applied to the visualization output image before save.
- **interval:** The interval that the visualization will be computed at.

Appendix D

Visualization Files

This appendix presents the format of the visualization files produced from the callbacks. The information can be useful for developers that want to extend the application or work directly with the visualization files instead of through the provided interface.

All examples shown are generated using a grayscale input image of shape (28, 28, 1). All arrays must be NumPy arrays.

D.1 Training Progress

Note: `progress` refers to the percentage of the training progress that the metrics occur at, for instance 0.1 is 10% of the first epoch, 1.5 is halfway in the second epoch, and so on.

- **Filename:** training_progress.txt
- **Format:** `progress loss accuracy`
- **Example:**

```
0.0 0.109375 2.315384864807129
0.0021321961620469083 0.140625 2.287837028503418
0.0042643923240938165 0.234375 2.2661919593811035
0.006396588486140725 0.1875 2.2656376361846924
0.008528784648187633 0.2890625 2.199150562286377
0.010660980810234541 0.25 2.2080719470977783
```

If validation is enabled, a separate text file for validation is created:

- **Filename:** training_progress_val.txt
- **Format:** `progress val_loss val_accuracy`
- **Example:**

```
1 0.97000 0.09730
2 0.97850 0.06922
```

D.2 Layer Activations

- **Filename:** layer_activations.pickle
- **Format:** `[(layer_name, array), ...]`
- **Example:**

```
[('Layer 1: conv2d_1', array(shape=(32, 26, 26), dtype=uint8),
 ('Layer 2: conv2d_2', array(shape=(32, 24, 24), dtype=uint8),
 ('Layer 3: max_pooling2d_1', array(shape=(32, 12, 12), dtype=uint8),
 ('Layer 6: dense_1', array(shape=(128,), dtype=uint8),
 ('Layer 8: dense_2', array(shape=(64,), dtype=uint8),
 ('Layer 9: dense_3', array(shape=(10,), dtype=uint8)]
```

D.3 Saliency Maps

- **Filename:** saliency_maps.pickle
- **Format:** `array`
- **Example:**

```
array(shape=(28, 28, 1), dtype=uint8)
```

D.4 Deconvolution Network

- **Filename:** deconvolution_network.pickle
- **Format:** `[(array, layer_name, feat_map_no), ...]`
- **Example:**

```
[(array(shape=(28, 28, 1), dtype=uint8), 'max_pooling2d_1', 0),
 (array(shape=(28, 28, 1), dtype=uint8), 'max_pooling2d_1', 10),
 (array(shape=(28, 28, 1), dtype=uint8), 'max_pooling2d_1', 22),
 (array(shape=(28, 28, 1), dtype=uint8), 'max_pooling2d_1', 3)]
```

D.5 Deep Visualization

- **Filename:** deep_visualization.pickle
- **Format:** `[(array, layer_name, unit_index, loss_value), ...]`
- **Example:**

```
[(array(shape=(28, 28, 1), dtype=uint8), 'dense_3', 8, 11761.4814453125),
 (array(shape=(28, 28, 1), dtype=uint8), 'dense_3', 9, 7111.62060546875),
 (array(shape=(28, 28, 1), dtype=uint8), 'dense_2', 8, 3238.24609375),
 (array(shape=(28, 28, 1), dtype=uint8), 'dense_1', 112, 0.0),
 (array(shape=(28, 28, 1), dtype=uint8), 'conv2d_2', (12, 12, 0), 0.0),
 (array(shape=(28, 28, 1), dtype=uint8), 'conv2d_2', (12, 12, 8), 0.0)]
```