

# CHALMERS



# Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification

Convolutional Neural Network Applications

Master's thesis in Complex Adaptive Systems

AUGUST VON HACHT



MASTER'S THESIS 2017:NN

**Implementation of Deep Feedforward  
Neural Network with CUDA backend  
for Efficient and Accurate Natural Image  
Classification**

Convolutional Neural Network Applications

AUGUST VON HACHT



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

Department of Physics  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Gothenburg, Sweden 2017

Implementation of Deep Feedforward Neural Network with CUDA backend for Ef-  
ficient and Accurate Natural Image Classification  
Convolutional Neural Network Applications  
AUGUST VON HACHT

© AUGUST VON HACHT, 2017.

Supervisor & Examiner: Mats Granath, Department of Physics

Master's Thesis 2017:NN  
Department of Physics  
Chalmers University of Technology  
SE-412 96 Gothenburg  
Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X  
Gothenburg, Sweden 2017

Implementation of Deep Feedforward Neural Network with CUDA backend for Efficient and Accurate Natural Image Classification  
Convolutional Neural Network Applications  
AUGUST VON HACHT  
Department of Physics  
Chalmers University of Technology

## Abstract

Recent advancements in techniques for constructing and training deep feedforward neural networks for classification tasks has enabled efficient training procedures leading to impressive results. This involves reducing overfitting due to over parameterized models, using an adaptive learning rate for avoiding exploding and vanishing gradients and symmetry breaking parameter initialization for efficient model optimization. Utilizing these techniques, this thesis concerns with the implementation of deep feedforward neural networks capable of efficient and accurate natural image classification. Four feedforward neural network models were constructed with the aim to classify tiny natural images from the CIFAR10 dataset. Having 3.274.634 trainable parameters for gray scale input and 4.259.274, 29.853.002 and 30.955.290 trainable parameters for rgb input, the training procedure utilizes a CUDA backend for efficient parameter optimization. The handwritten digits were classified with 97.31% accuracy and the tiny natural images were classified, using the best model, with 72.88% accuracy.

Keywords: Deep feedforward neural networks, Convolutional neural networks, CUDA, Natural Image classification



# Acknowledgements

I would like to thank my supervisor and examiner Mats Granath for the support and encouragement in pursuing this thesis work.

I would also like to thank Chalmers Centre for Computational Science and Engineering for allowing me to utilize their cluster computer as a basis for the demanding computations. Thanks for a helpful support service center and a fruitful introduction to the cluster environment.

Without the MNIST and CIFAR10 dataset this work would not be possible and I would therefore like to thank the maintainers of respective dataset.

Finally, I would like to take the opportunity to express my love and gratitude towards my family for their endless support during the writing of this thesis.

August von Hacht, Gothenburg, June 2017





# Contents

<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Deep feedforward neural networks . . . . .	2
<b>2 Theory</b>	<b>5</b>
2.1 Supervised learning for optimizing deep neural networks . . . . .	5
2.2 Optimization through gradient based learning . . . . .	6
2.2.1 Backpropagation in feedforward neural networks . . . . .	6
2.2.2 Gradient based learning rule . . . . .	8
2.2.3 Adaptive moment estimation . . . . .	8
2.3 Feedforward neural network layer components . . . . .	9
2.3.1 Fully connected layer . . . . .	9
2.3.2 Convolutional layer . . . . .	10
2.3.3 Maxpooling layer . . . . .	11
2.4 Activation function . . . . .	13
2.4.1 Rectified linear unit . . . . .	13
2.5 Regularization . . . . .	13
2.5.1 Weight decay . . . . .	14
2.5.2 Dropout . . . . .	14
2.6 Weight initialization . . . . .	15
<b>3 Methods</b>	<b>17</b>
3.1 Implementation of deep feedforward neural network . . . . .	17
3.1.1 Data representation . . . . .	17
3.1.2 Tensor representation as multidimensional array . . . . .	17
3.1.3 Convolution as matrix multiplication . . . . .	18
3.1.4 Convolution output to fully connected input conversion for matrix multiplication . . . . .	19
3.1.5 The computational graph . . . . .	19
3.1.6 Context object pattern . . . . .	20
3.2 Efficient matrix multiplications . . . . .	21
3.2.1 OpenBLAS . . . . .	21
3.2.2 cuDNN and cuBLAS . . . . .	21

3.3	Verify computations through gradient check . . . . .	22
3.4	Implementation verification using the MNIST dataset . . . . .	23
3.4.1	Pre-processing . . . . .	24
3.4.2	Network architecture . . . . .	24
3.4.3	Hyperparameters . . . . .	25
3.5	Challenging the CIFAR10 dataset . . . . .	25
3.5.1	Pre-processing . . . . .	26
3.5.2	Network architectures . . . . .	26
3.5.3	Hyperparameters . . . . .	28
3.6	Neural network architecture evaluations . . . . .	28
3.6.1	Loss . . . . .	28
3.6.2	Accuracy . . . . .	28
3.6.3	Confusion matrix . . . . .	28
3.6.4	Weight visualization . . . . .	28
<b>4</b>	<b>Results</b>	<b>31</b>
4.1	MNIST classification . . . . .	31
4.1.1	Discussion . . . . .	33
4.2	CIFAR10 classification . . . . .	33
4.2.1	Model A . . . . .	34
4.2.2	Model B . . . . .	36
4.2.3	Model C . . . . .	38
4.2.4	Discussion . . . . .	40
4.2.4.1	Model A . . . . .	40
4.2.4.2	Model B . . . . .	41
4.2.4.3	Model C . . . . .	41
4.2.4.4	Comparison with the state-of-the-art . . . . .	43
4.2.5	Pre-processing . . . . .	43
4.2.6	Optimizing the neural network architectures . . . . .	44
<b>5</b>	<b>Conclusions</b>	<b>45</b>
5.1	Future work . . . . .	45
	<b>Bibliography</b>	<b>47</b>
<b>A</b>	<b>Details to formulas in theory section</b>	<b>I</b>
A.1	Negative log-likelihood gradient . . . . .	I
A.2	Backpropagation: Fully connected layer . . . . .	I
A.3	Backpropagation: Convolutional layer . . . . .	III
A.4	Gaussian weight initialization . . . . .	IV
<b>B</b>	<b>Two hyperparameter trials</b>	<b>VII</b>
B.0.1	Model A trial one . . . . .	VII
B.0.2	Model A trial two . . . . .	IX

# List of Figures

2.1	Negative log likelihood . . . . .	6
3.1	Two dimensional row-major ordering . . . . .	18
3.2	Computational graph . . . . .	20
3.3	MNIST dataset . . . . .	23
3.4	MNIST dataset distribution . . . . .	24
3.5	CIFAR10 dataset . . . . .	25
3.6	CIFAR10 dataset distribution . . . . .	26
4.1	MNIST classification results . . . . .	31
4.2	MNIST learned weights . . . . .	32
4.3	MNIST confusion matrix . . . . .	33
4.4	CIFAR10 model A classification results . . . . .	34
4.5	CIFAR10 Model A accuracy . . . . .	34
4.6	CIFAR10 model A learned weights . . . . .	35
4.7	CIFAR10 Model A confusion matrix . . . . .	36
4.8	CIFAR10 model B loss . . . . .	36
4.9	CIFAR10 Model B accuracy . . . . .	37
4.10	CIFAR10 Model B learned weights . . . . .	37
4.11	CIFAR10 Model B confusion matrix . . . . .	38
4.12	CIFAR10 model C classification results . . . . .	38
4.13	CIFAR10 Model C accuracy . . . . .	39
4.14	CIFAR10 Model C learned weights . . . . .	39
4.15	CIFAR10 Model C confusion matrix . . . . .	40
4.16	Gabor filters . . . . .	41
4.17	von Neumann neighborhood . . . . .	42



# List of Tables

3.1	Graphics card specifications . . . . .	22
3.2	MNIST architecture . . . . .	24
3.3	CIFAR10 model A . . . . .	26
3.4	CIFAR10 model B . . . . .	27
3.5	CIFAR10 model C . . . . .	27
4.1	MNIST hyperparameters . . . . .	31
4.2	CIFAR10 model A hyperparameters . . . . .	34
4.3	CIFAR10 model B hyperparameters . . . . .	36
4.4	CIFAR10 model C hyperparameters . . . . .	38



# 1

## Introduction

### 1.1 Motivation

The motivation behind this thesis stems from a guest lecture about the Never-Ending language learner (NELL) [1] and the recent advancements in the field of deep learning [2]. The NELL project aims at learning a set of relations between nouns through reading text. As basic understanding of context usually requires additional information inferred from prior knowledge, a system which learns a set of prior knowledges, which has shown to be useful in interpreting text, would be interesting to experiment with within the context of images and videos. With the recent advancement in image recognition through deep learning such a pipeline seems obtainable for interpreting basic context in images and videos. Especially, since most deep learning networks generates a short vector of a complex representation, which can easily and efficiently be stored in a database. However, it is unclear whether the recent results are due to development of the deep learning algorithms or rather a consequence of fine tuning, computational power and data specific techniques. The recent impressive results are often attributed to the increase in computational power and data, allowing for deeper neural networks to be tested. The computational power comes from utilizing the graphics processing unit (GPU) to perform large parallel computations which drastically reduces the required optimization time.

It is in the interest of this thesis to delve into the implementation and details utilized for learning to classify natural images. Gaining deeper insights in the techniques behind the successful results may help to improve upon the current knowledge in deep feedforward neural networks and their applications. Setting up a neural network to automatically learn fundamental representations between nouns simplifies the tedious work of doing so manually. This thesis is therefore concerned with the following tasks

- Implement a deep feedforward neural network capable of natural image classification.
- Utilize the graphics processing unit to speed up the optimization process.

Using this implementation the follow research questions are studied

- What is a good way to construct a deep feedforward neural network for natural image classification?
- Do these neural networks, using only mean subtraction on the data, obtain a good accuracy?
- Does a deep feedforward neural network learn properties that generalize?

With good accuracy means high accuracy  $> 90\%$ .

Further details into how to construct a pipeline for learning relations in images and video is beyond the scope of this thesis and will not be further addressed. The deep feedforward neural network to investigate and implement is the convolutional neural network and so this thesis intends its applications. The natural images to apply the developed deep learning framework to is the CIFAR10 dataset [3]. The dataset contains natural images of 10 mutually exclusive classes.

## 1.2 Deep feedforward neural networks

Deep feedforward neural networks are directed acyclic graphs (DAG) consisting of layers of neurons, further referred to as computational units. The network is feedforward due to having layer connections which follows a DAGs topological ordering. A layer in the network is defined as a set of computational units which together with input  $\mathbf{x}$  computes some function  $\vec{f}(\mathbf{x}; \boldsymbol{\theta})$ , where  $\boldsymbol{\theta}$  is a trainable parameter set belonging to a layer (bold symbol and arrow indicates vector). A feedforward network is constructed using an input layer, hidden layers and an output layer. Given a network with  $N$  hidden layers this network computes a mapping  $\mathbf{y}$  of the input  $\mathbf{x}$  as

$$\mathbf{y} = \vec{f}^{(N+1)}(\dots(\vec{f}^{(2)}(\vec{f}^{(1)}(\mathbf{x}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_2))\dots; \boldsymbol{\theta}_{N+1}) \quad (1.1)$$

This construction gives a deep feedforward neural network the ability to approximate a desired mapping through combining multiple layers which each computes a function  $\vec{f}^{(i)}$  on their respective input and by tuning their parameters  $\boldsymbol{\theta}_i$ .

These network ideas of connected neurons stems from inspiration from the fields of neuroscience. Much of these neural network characteristics originates from simplified models of how the human brain has been observed to behave on the scale of the biological neuron, e.g. the McCulloch-Pitts neuron [4]. However, these neuron models omit many complications such as; biological neurons produces time series of spikes as a response of continuous input, asynchronous updating of individual neurons and nonlinear summations. Therefore, feedforward neural networks are better thought of as a design aimed at achieving statistical generalization which has drawn insight from the biological neurons behavior.

Despite these simplifications a neural network with a single hidden layer having  $m$  neurons can approximate a wide variety of continuous functions on compact subsets of  $\mathbf{R}^n$ , under mild conditions on the activation functions [5, 6]. These theoretical results only shows whether any particular function is computable or not, to which the answer is always yes. However, for practical usage the question of what is a good way to construct a neural network to compute a particular mapping has shown to be difficult to answer. While in principle any well enough approximated mapping can be found using a shallow neural network there are many practical reasons for using deep neural networks. For instance in image recognition, learning not only



the individual pixel values but also the structures of which a set of pixels can form helps to understand complex concepts by forming hierarchies of abstractions, e.g. geometrical shapes, and thus gaining a 'deeper' representation of the data.



# 2

## Theory

This chapter outlines the deep feedforward neural network components used to construct a deep learning model for image recognition of natural images.

### 2.1 Supervised learning for optimizing deep neural networks

Supervised learning is the task of inferring a function from supervised training examples. Given a set of  $N$  data points  $\{(x_1, y_1), \dots, (x_N, y_N)\}$  where  $x_i \in \mathbf{R}^D$  is a vector representation of an instance of a class and  $y_i \in 1 \dots K$  is a class label, a learning algorithm seeks a mapping  $f : \mathbf{R}^D \rightarrow \mathbf{R}^K$ . It is common to represent  $f$  using a score function  $g : \mathbf{R}^D \times \mathbf{R}^K \rightarrow \mathbb{R}$  such that the label  $y$  from the set of class labels having the highest score is returned

$$\operatorname{argmax}_y g(x, y) \tag{2.1}$$

While the score function simply makes a prediction of the class label based on the neural network output, improvements on the parameterized computation resulting in that output can be made through optimization. This is done via measuring the error rate of the current prediction, before passing it through the score function, using a loss function  $L : \mathbf{R} \times \mathbf{R}^K \rightarrow \mathbb{R}$ . For a data point  $(x_i, y_i)$  the loss of predicting output  $\hat{\mathbf{y}}$  is  $L(y_i, \hat{\mathbf{y}})$ . The computation, in this case a forward pass through the deep feedforward neural network, can therefore learn its parameters through minimizing the loss function as this corresponds to optimizing the fit. Thus, the neural network learns what a representation  $x$  corresponds to among the classes  $y$  through inferring such a mapping from training data.

In many applications it is convenient for the output of the neural network, before it is forwarded through the score function, to have some natural interpretation as this make for an easier interpretation of the score function. In image classification tasks converting the output into a probability distribution over the class labels gives the interpretation of the highest scoring element as the label which the network believes is most probable to associated with the image.

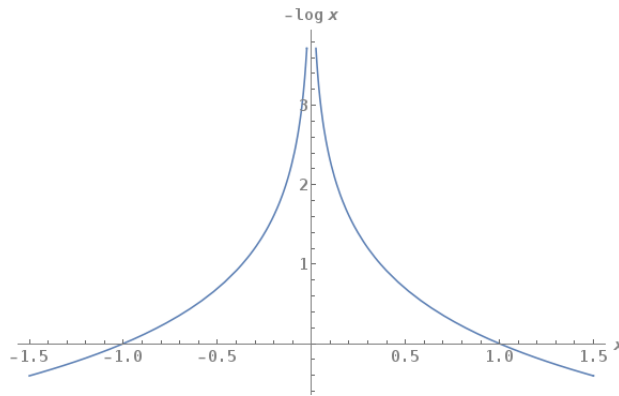
The transformation from the network output to a probability distribution is done

via the softmax function

$$p(\hat{\mathbf{y}})_k = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} \quad (2.2)$$

As the neural network outputs a probability the error rate of this belief is measured using the negative log likelihood

$$L(y_i, \hat{\mathbf{y}}) = -\log(p(\hat{\mathbf{y}})_{y_i}) \quad (2.3)$$



**Figure 2.1:** Depending on the element at the index  $y_i$  in the output vector, the negative log likelihood returns a loss reflecting its error rate.

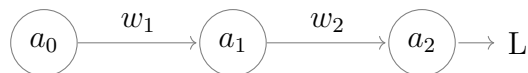
The kink at  $p(\hat{\mathbf{y}})_{y_i} = 0$  is counteracted by adding a small numerical constant which does not affect the result during normal operation.

## 2.2 Optimization through gradient based learning

Given a poor prediction by the network, improving the fit can be done by tuning the parameters  $\theta$ . Due to the non-convex optimization problem, an iterative gradient-based optimization scheme over the high dimensional parameter space is required.

### 2.2.1 Backpropagation in feedforward neural networks

Assume the following simple feedforward neural network



where  $a_j = f(z_j)$  is the activation function  $f$  applied to the affine transformation  $z_j = w_j a_{j-1}$  and  $L$  is the loss. A small change in  $\Delta w_1$  will cause a change to  $a_1$  by

$$\Delta a_1 \approx \frac{\partial a_1}{\partial w_1} \Delta w_1 = f'(z_1) a_0 \Delta w_1 \quad (2.4)$$

this will in turn change  $z_2$  by

$$\Delta z_2 \approx \frac{\partial z_2}{\partial a_1} \Delta a_1 = w_2 \Delta a_1 \quad (2.5)$$

which will trigger a change in  $a_2$  by

$$\Delta a_2 \approx \frac{\partial a_2}{\partial z_2} \Delta z_2 = f'(z_2) \Delta z_2 \quad (2.6)$$

finally, this change causes

$$\Delta L \approx \frac{\partial L}{\partial a_2} \Delta a_2 \quad (2.7)$$

therefore, in this case, a change in the parameter  $w_1$  results in a change in  $L$

$$\frac{\Delta L}{\Delta w_1} = f'(z_1) a_0 w_2 f'(z_2) \frac{\partial L}{\partial a_2} \quad (2.8)$$

Thus, given an error  $\delta_2 = \frac{\partial L}{\partial a_2}$  at the output, running this procedure in reverse gives the amount by which a parameter  $w_j$  should be changed in order to achieve a smaller error. Usually deep feedforward neural networks contain many layers with different layer structures and weight connections, which is why the above procedure scales poorly. Instead, given the intermediate changes or errors at a layer  $l$ , defining the backpropagation procedure through the individual layers gives the same procedure but in an iterative form. The error is therefore defined using the iterative form

$$\delta_j^l \equiv \frac{\partial L}{\partial z_j^l} \quad (2.9)$$

which instead implicitly holds all the errors from the previous layers. Defining the error using the affine transformation instead of the activation function is for convenience in the appendix derivations. The only change is having the derivative of the activation function  $f'(z_j)$  showing up in the layer-specific backpropagation equations.

The objective in optimizing the neural network is to minimize the loss function  $L$  with respect to its parameters, which corresponds to optimizing the fit. This requires non-convex optimization, which is done via stochastic gradient descent or in other words at each time step reducing the loss by taking a step in the direction of the steepest descent in parameter space. This, as seen by eq. (2.8), requires the output gradient, which given a prediction  $\hat{\mathbf{y}}$  and using the negative log likelihood eq. (2.3) is

$$\frac{\partial L(y_i, \hat{\mathbf{y}})}{\partial \hat{y}_k} = \begin{cases} p(\hat{\mathbf{y}})_k - 1 & \text{if } k = y_i \\ p(\hat{\mathbf{y}})_k & \text{otherwise} \end{cases} \quad (2.10)$$

where  $y_i$  is the correct label associated with the input which results in the prediction  $\hat{\mathbf{y}}$  (details are in appendix A.1).

### 2.2.2 Gradient based learning rule

The stochastic gradient descent update rule for an individual neuron parameter at time  $t$  is

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \frac{\partial L_t}{\partial \theta_t} \quad (2.11)$$

where  $\eta_t$  is the step size at time  $t$ , commonly called the learning rate, and  $\frac{\partial L}{\partial \theta}$  is the error in  $\theta$ , defined as a chain of gradients, seen from eq. (2.8).

In deep feedforward neural networks the gradients in the update rule has shown to be problematic due to long chains of gradients. Problems such as the vanishing gradient problem where the gradients in the early layers are close to zero due to a parameter somewhere in the later layers gets stuck when having to moving across a saddle point, which causes the learning to stagnate for earlier layers. Therefore, it is proposed to use an adaptive learning rate which bounds the gradients by the learning rate, making the learning independent of these chain lengths. Additionally, it also accounts for sparsity by updating infrequent occurrences stronger and scaling down frequent occurring parameter updates. This has show to give a greater robustness of the update rule when training to classify images.

### 2.2.3 Adaptive moment estimation

Adaptive moment estimation [7] maintains a history over an exponentially decaying average of past gradients

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L_t}{\partial \theta_t} \quad (2.12)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left( \frac{\partial L_t}{\partial \theta_t} \right)^2 \quad (2.13)$$

where  $m_t$  and  $v_t$  are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. The authors (Kingma and Ba) found that the zero initialization biased the estimates toward zero and therefore counteracted this by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (2.14)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (2.15)$$

thereafter the update rule eq. (2.11) is instead

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \quad (2.16)$$

where  $\epsilon$  is a small numerical constant added to avoid division by zero. The step  $\theta_{t+1} - \theta_t$  taken in parameter space at time  $t$  is bounded by

$$\begin{aligned} |\Delta_t| &\leq \eta_t \left( \frac{1 - \beta_1}{\sqrt{1 - \beta_2}} \right) && \text{if } 1 - \beta_1 > \sqrt{1 - \beta_2} \\ |\Delta_t| &\leq \eta_t && \text{otherwise} \end{aligned} \quad (2.17)$$

For further details of the bounds, see the authors paper (Kingma and Ba). This update rule prevents the gradients from either exploding or vanishing, independently of the neural network depth. A modification of this learning rule is implemented and used. The modified version introduces weight decay and is presented in a section 2.5.1.

## 2.3 Feedforward neural network layer components

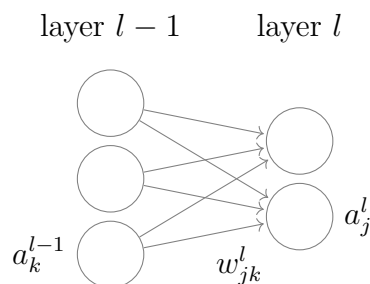
The feedforward neural network used contains two types of feedforward structures. The first structure is the combination of one or more convolutional layer followed by a maxpooling layer. Multiple such structures can be combined forming a ConvNet structure which aims at simplifying an image by measuring activations from affine transformations. The other structure is a combination of one or more fully connected layers which aims at classifying these representations.

### 2.3.1 Fully connected layer

The fully connected layer is parametrized by weights and a bias. Similar to the McCulloch-Pitts neuron the neurons in a fully connected layer forwards an activation function  $f$  applied to the weighted sum over the inputs

$$a_j^l = f \left( \sum_k w_{jk}^l a_k^{l-1} + b_j^l \right) \quad (2.18)$$

where the activation  $a_j^l$  of the  $j^{\text{th}}$  neuron in the  $l^{\text{th}}$  layer is related to the  $k^{\text{th}}$  neuron in the  $(l-1)^{\text{th}}$  layer via the weight  $w_{jk}^l$  shown in the following diagram



The bias term is usually thought of as an extra neuron connected to the  $l^{\text{th}}$  layer via a weight of value one. The input to the activation function  $f$  is referred to as the affine transformation of the input

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad (2.19)$$

and given a parameters setting resulting in an error  $\Delta z_j^l$  this is backpropagated as

$$\delta_k^{l-1} = \sum_j w_{jk}^l \delta_j^l f'(z_k^{l-1}) \quad (2.20)$$

Now given an error at a neuron the weight and bias gradients are given as

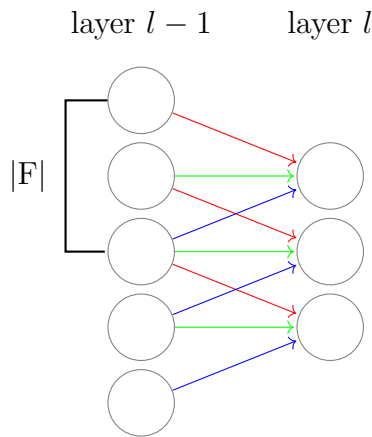
$$\frac{\partial L}{\partial b_j^l} = \delta_j^l \quad (2.21)$$

$$\frac{\partial L}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (2.22)$$

with the details in appendix (A.2).

### 2.3.2 Convolutional layer

A convolutional layer is also parametrized by weights and a bias but introduces additional parameters through weight sharing. The weight sharing principle is shown in a one dimensional case using a color scheme to showcase the same weight occurrences



Unlike the fully connected layer where each neuron connects to the next layer via a unique weight, the convolution layer use a set of weights called filters  $F$  which is applied multiple times onto various spatial locations of the input  $I$ . As a result, the number of weights in the one dimensional case remains constantly equal to the filter size instead of  $O(nm)$ , where  $n$  is the number of neurons in layer  $l$  and  $m$  is the number of neuron is layer  $l - 1$ , as in the fully connected case.

The weight sharing introduces additional parameters called stride and padding. The stride determines the step size by which the filters moves over the input, e.g. in the diagram the stride equals to one. The padding is used to fill out the input's spatial domain such that the convolution is defined at the borders for a certain filter size and stride. These parameters are not learnable and instead used to define the spatial output domain which is related to the spatial input domain through

$$n_i^l = \left\lfloor \frac{n_i^{l-1} - f_i + 2p_i}{s_i} \right\rfloor + 1 \quad (2.23)$$

where  $i$  denote the domain by which the filters acts, e.g. width or height of an image. Note that the output domain before the floor function is preferably integer valued. It is also preferred to have  $n_i^l = n_i^{l-1} \quad \forall i$  since this makes the filters occur



at the same spatial locations in both the backward and forward pass.

The two operations performed in a forward and backward pass of a convolutional layer are the discrete cross-correlation

$$C_{jk}^{\text{cross}} = \sum_{j'} \sum_{k'} I_{j+j', k+k'} F_{j'k'} \quad (2.24)$$

and discrete convolution

$$C_{jk}^{\text{conv}} = \sum_{j'} \sum_{k'} I_{j-j', k-k'} F_{j'k'} \quad (2.25)$$

which are related through a flipping operation applied onto the filters. During a forward pass through the layer each output activation is formed by

$$a_{jk}^l = f \left( w_{jk}^l * a_{jk}^{l-1} + b_{jk}^l \right) \quad (2.26)$$

$$= f \left( \sum_{j'} \sum_{k'} w_{j'k'}^l a_{j-j', k-k'}^{l-1} + b_{jk}^l \right) \quad (2.27)$$

The input to the activation function  $f$  is again an affine transformation denoted by

$$z_{jk}^l = \sum_{j'} \sum_{k'} w_{j'k'}^l a_{j-j', k-k'}^{l-1} + b_{jk}^l \quad (2.28)$$

Now given an error  $\Delta z_{jk}^l$  this is propagated backwards according to

$$\delta_{jk}^{l-1} = \delta_{jk}^l * \text{rot}_{180^\circ}(w_{jk}^l) f'(z_{jk}^l) \quad (2.29)$$

Looking back at the one dimensional case in the diagram, the color scheme shows that when forward propagating a value in layer  $l-1$  to layer  $l$  this requires traversing the weight connections according to red, green and blue order. When backward propagating an error in layer  $l$  to layer  $l-1$  this requires traversing the weights according to blue, green and red order. This is the reason for rotating the filters in eq. (2.29). Given an error at each neuron in a convolutional layer the weights and bias gradients are

$$\frac{\partial L}{\partial w_{jk}^l} = \delta_{jk}^l * f(\text{rot}_{180^\circ}(z_{jk}^{l-1})) \quad (2.30)$$

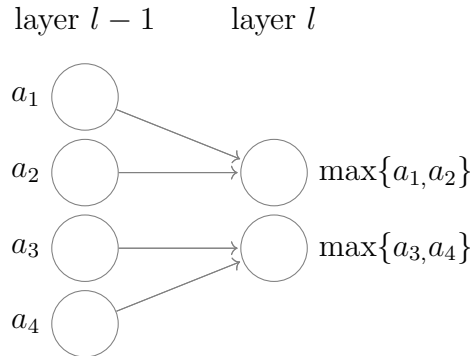
$$\frac{\partial L}{\partial b_{jk}^l} = \delta_{jk}^l \quad (2.31)$$

with the details in appendix (A.3).

### 2.3.3 Maxpooling layer

A maxpooling layer summarizes the output from a convolutional layer by grouping output regions into a single representative candidate. This layer therefore does not require any trainable parameters but has to describe the choice of the region used

by the max operator. A region is a two dimensional rectangular neighborhood with side lengths  $r_x$  and  $r_y$  and when applied to the input space encloses some subset of activation values. Multiple regions are spread either jointly or disjointly across the input space which is decided by a striding and padding parameter. If the stride size along a spatial domain is larger than the rectangular side length in that domain then the regions won't overlap otherwise they will. The output's spatial domain is obtained through eq. (2.23), but with  $f_i \rightarrow r_i$ . Using a one dimensional case with stride equals to two and padding zero results in the following diagram



The particular form of weight sharing used in the convolutional layer has the property of being equivariant to translation. Equivariance refers to the property of changing input results in the output changing similarly. This means that first translating and then convolving is equal to first convolving and then translating, assuming the translation keeps the object within the input domain,

$$[[L_t F] * I](x) = \sum_y F(y - t)I(x - y) \quad (2.32)$$

$$= \sum_{y'} F(y')I(x - (y' + t)) \quad (2.33)$$

$$= \sum_{y'} F(y')I(x - y' - t) \quad (2.34)$$

$$= [L_t [F * I]](x) \quad (2.35)$$

By summarizing the output from a convolutional layer using a maxpooling layer the resulting representation of the activations is made approximately invariant to small uniform translations in the input. If a small uniform translation in the input occurs then this is canceled out by the max operator since the winning activations maintain their relative positions. For image classification tasks this makes such a layer combination invariant to where in the image an object of the same class is positioned spatially.

The forward pass through a maxpooling layer is

$$a_{jk}^l = \max \left( \left\{ a_{js_y - p_y, ks_x - p_x}^{l-1}, a_{js_y - p_y, ks_x + 1 - p_x}^{l-1}, \dots, a_{js_y + r_y - p_y, ks_x + r_x - p_x}^{l-1} \right\} \right) \quad (2.36)$$

where  $s_x, s_y$  are the striding parameters and  $p_x, p_y$  are the padding parameters of respective domains.

Since there are no trainable parameters the errors propagated backwards from the next layer is simply back propagated to the previous layer as is. However, for each region only the winning activation obtains the error since the other activations did not participate in the computation of the output mapping

$$\delta_{jk}^{l-1} = \begin{cases} \delta_{jk}^l & \text{if } a_{jk}^{l-1} \text{ was max} \\ 0 & \text{otherwise} \end{cases} \quad (2.37)$$

## 2.4 Activation function

The activation function is used to restrict a neurons output within a certain range and to expand the mapping  $f$  beyond linear regression. The consequence of using an activation function is however that a non-constant derivative shows up in the gradients. Confining the output range to  $[0, 1]$  using the sigmoid function or to  $[-1, 1]$  using the hyperbolic tangent is useful for stabilizing the gradients during optimization. However, near these borders the gradients vanishes which is problematic for deep neural networks since this will stagnate the learning for a neuron in a layer and all neurons in earlier layers sharing connections to the output via this neuron, as seen by eq. (2.8). Therefore, the rectified linear unit is used for optimizing deeper neural networks.

### 2.4.1 Rectified linear unit

The rectified linear unit (ReLU) [8] is defined as

$$f(z) = \max(0, z) \quad (2.38)$$

where

$$f'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.39)$$

which is a non-saturating non-linearity that has shown to be efficient in optimizing deep neural networks.

## 2.5 Regularization

Deep feedforward neural networks are inherently overparametrized models aimed at optimizing a fit to data points. Due to having more parameters then required the neural network has to be regularized in order to avoid overfitting on non-generic structures, such as noise. It is typically the case that the number of data points available are orders of magnitude less than the number of parameters used by the neural network. Therefore, two regularization schemes are adopted where the first technique modifies the loss function whereas the second technique modifies the neural network architecture.

### 2.5.1 Weight decay

When trying to fit a high dimensional hyperplane to a number of data points the probability of a unique optimal fit is very unlikely. Any infinitesimal change to any of the hyperplane parameters has to worsen the fit which together with a finite number of data points makes this certainly improbable. It is therefore a reasonable assumption that there exists multiple sets of these parameters which optimizes the fit. Moreover, when optimizing with a subset of the data various sets of parameters might improve the fit due to utilizing local structures. A technique for selecting a specific set of parameters and avoiding overfitting on local structures in the data so that the generalization capabilities of the network improves is to introduce weight decay.

Weight decay adds an additional term to the loss function

$$L(y_i, \hat{\mathbf{y}}) = -\log(p(\hat{\mathbf{y}})_{y_i}) + \frac{\lambda}{2} \sum_w w^2 \quad (2.40)$$

where the sum is over all weights in the neural network and  $\lambda$  is the regularization term. This  $L2$ -regularization tells the network to prefer smaller weights. Large weight are only allowed if they make a large improvement to the loss function. The reason it is called weight decay is because the regularization term introduces to the gradient of the loss function, with respect to the weight, a decay term which equals  $\lambda w$ . The stochastic gradient descent update rule using adaptive moment estimation is therefore modified to

$$w_{t+1} \leftarrow (1 - \eta_t \lambda) w_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \quad (2.41)$$

### 2.5.2 Dropout

Dropout is a regularization technique used for reducing over fitting by approximating many different neural network architectures [9]. During optimization, a neurons activation is randomly dropped temporarily by cutting off the weight connection to the next layer. Over many training examples this corresponds to optimizing a set of approximately different neural networks since having  $n$  units in a layer can have  $2^n$  different arrangements of connections to the next layer. When predicting, having each unit active, a weight connection is approximated by the expected value of being active times the value to propagate forward.

Dropout can be modeled as a simple layer with non-trainable weights of value one. It has one non-trainable parameter  $p$ , where  $1 - p$  is the probability of setting a weight to zero which effectively cut off the connection to the next layer. A forward pass during optimization is

$$r_j^l \sim \text{Bernoulli}(p) \quad (2.42)$$

$$\hat{y}_j^l = r_j^l y_j^l \quad (2.43)$$

and

$$\hat{y}_j^l = E[r_j^l]y_j^l \quad (2.44)$$

$$= py_j^l \quad (2.45)$$

when predicting. The backward pass simply backpropagates an error through a dropout layer depending on what  $r$  was in the forward pass

$$\delta_j^{l-1} = \begin{cases} \delta_j^l & \text{if } r_j^l \text{ was 1} \\ 0 & \text{otherwise} \end{cases} \quad (2.46)$$

## 2.6 Weight initialization

The weight initialization has shown to be crucial for optimizing deep feedforward neural networks [10, 11]. Without proper weight initialization the variation of the input signal and the error signal is exponentially scaled, which has shown to cause learning to diverge or stall immediately after initialization. In the early stages of optimization it has also been shown to benefit the neural network if the output variation is uniformly spread across the outputs in any layer. This enhances learning through keeping larger parts of the neural network active with backpropagated non-zero errors during the initial optimization steps.

The variance in the input using ReLU activation propagates forward through the network as

$$\text{Var}[z^l] = \text{Var}[z^1] \left( \prod_2^L \frac{1}{2} n^l \text{Var}[w^l] \right) \quad (2.47)$$

and backwards as

$$\text{Var}[\delta^2] = \text{Var}[\delta^{L+1}] \left( \prod_2^L \frac{1}{2} \hat{n}^l \text{Var}[w^l] \right) \quad (2.48)$$

where  $n^l$  are the number of terms in the affine transformation of the forward pass,  $\hat{n}^l$  are the number of terms in the affine transformation of the backward pass. A proper weight initialization should avoid scaling the input signal exponentially, which leads to the conditions

$$\frac{1}{2} n^l \text{Var}[w^l] = 1, \quad \forall l \quad (2.49)$$

$$\frac{1}{2} \hat{n}^l \text{Var}[w^l] = 1, \quad \forall l \quad (2.50)$$

It is sufficient to fulfill either of the two conditions since this leads to the other condition not resulting in a diminishing number in many common feedforward neural networks. The first condition is fulfilled by a zero-mean Gaussian distribution whose standard deviation is  $\sqrt{\frac{2}{n^l}}$  (details in A.4).



# 3

## Methods

The chapter outlines the consideration made during the implementation of the components outlined in the theory section, how the final implementation is verified, how it is tested and how the results are evaluated.

### 3.1 Implementation of deep feedforward neural network

The mapping eq. (1.1) is operating multiple times with various functions onto input which initially is represented by a natural image. To efficiently propagate the data throughout the neural network multiple consideration regarding the data structure and data propagation was made. These are presented below.

#### 3.1.1 Data representation

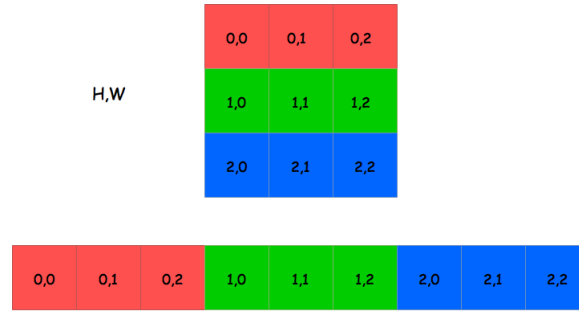
The neural network uses a tensor of fourth order to represent any type of data, e.g. filters, biases, input images, weights etc. To describe the tensor a shape is used which follows the row-major ordering meaning the rightmost index varies fastest. A shape has the following structure

$$\text{shape} = [N, C, H, W] \tag{3.1}$$

$N$  can be the number of images feed to the neural network in a forward pass or the number of filters used by a convolution layer.  $C$  is used to represent data channels e.g. grayscale or RGB color channels. The last two dimensions are the data height and width e.g. filter sizes or image sizes.

#### 3.1.2 Tensor representation as multidimensional array

An efficient representation of a tensor is the  $n$ -dimensional array which maps a certain index to a number on the line, the following figure shows how this line is formed



**Figure 3.1:** Example of using row-major ordering for representing a tensor of order two with a 2-dimensional array. In a 4-dimensional array each row in a matrix of width  $W$  and height  $H$  is stacked after each other, i.e.  $H$  stacks of  $W$  elements each. Thereafter, the next matrix corresponding to the next channel  $C$ , is converted similarly to an array and is stacked afterwards. This is repeated for each number  $N$ .

The general formula for finding a certain tensor element on the line in an  $n$ -dimensional array using row-major ordering is

$$\text{line position} = \sum_{i=1}^n \left( \prod_{j=1}^{i-1} N_j \right) n_i \quad (3.2)$$

In this case,  $n = 4$  and each  $N_j \in \{N, C, H, W\}$  which is given by the tensor shape. Many efficient matrix multiplication libraries requires a  $n$ -dimensional array with row-major ordering, so for compatibility any data is stored in this format, but used as if it was in the tensor format.

### 3.1.3 Convolution as matrix multiplication

Using the data representation eq. (3.1), an input to the convolutional layer has the shape  $[1, C, H, W]$ . A filter has the shape  $[K, C, F_h, F_w]$  where  $K$  is the number of filters and  $F$  is their width and height. In the affine transformation in the forward pass eq. (2.28) each filter is applied multiple times over the input. The number of locations are founding using the filter size  $F_h, F_w$ , the stride and the padding according to eq. (2.23) which is assumed to be  $VL$  for vertical locations and  $HL$  for horizontal locations. By converting the input to have the shape  $[1, 1, C \times F_h \times F_w, VL \times HL] = [C \times F_h \times F_w, VL \times HL]$  each column is effectively the locations which a filter is to multiply with, which requires each of those input values to be copied over to respective column position. This is done using an image to column operations. The filters are converted to  $[1, 1, K, C \times F_h \times F_w] = [K, C \times F_h \times F_w]$  and the multiplication follows as  $[K, C \times F_h \times F_w] \times [C \times F_h \times F_w, VL \times HL] = [K, VL \times HL] = [1, 1, K, VL \times HL]$ . Using a column to image operations the input can be restored to  $[1, K, VL, HL]$ .

The affine transformation in the backward pass eq. (2.29) follows by taking the errors of shape  $[1, K, VL, HL]$  and convert to  $[1, 1, K \times F_h \times F_w, H \times W] = [K \times F_h \times F_w, H \times W]$  using the same filter size, stride, padding and image to column operation. This is then multiplied by the weights flipped  $90^\circ$  horizontally and then  $90^\circ$  vertically; iterating through in row-major order would give the elements in backwards



order (recall the one dimensional color scheme diagram). These filters are converted into rows  $[1, 1, C, K \times F_h \times F_w] = [C, K \times F_h \times F_w]$ . The multiplication follows as  $[C, K \times F_h \times F_w] \times [K \times F_h \times F_w, H \times W] = [C, H \times W] = [1, 1, C, H \times W]$ . Using the inverse column to image operation the input shape is recovered as  $[1, C, H, W]$ . Given the error for each neuron in a layer the filter and bias gradients are found using eq. (2.34) and eq. (2.35) in their matrix multiplication forms.

The above matrix multiplication does not correspond to a proper convolution. The proper convolution requires the weights to be flipped in the forward pass and additionally flipped in the backward pass, just to restore the weights. To reduce this overhead one can not flip the weights in the forward pass and effectively perform cross-correlation. The backward pass then becomes a proper convolution. This is done on the CPU. However, an optimized network using this set of weights rather than the set obtain from convolving does yield the same loss. This because the neural network optimization does not depend on the order of the weights, hence the non-convex property of the parameter landscape. But using convolution with flipped weights corresponds to using cross-correlation without flipping the weights, so it is sufficient to use cross-correlation in both the forward and backward case. This is done on the GPU.

### 3.1.4 Convolution output to fully connected input conversion for matrix multiplication

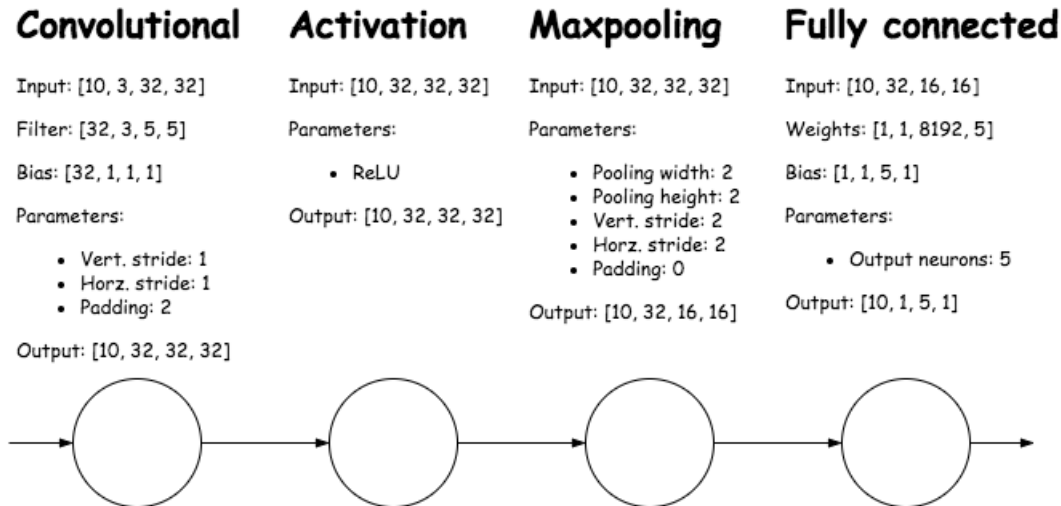
With the data representation used, an input from a convolution  $[1, C, H, W]$  is converted to a vector of shape  $[1, 1, C \times H \times W, 1] = [C \times H \times W, 1]$  with no additional computational cost, since it can be done by changing the constants  $N_j$  in eq. (3.2). The weights shape is  $[1, 1, D, C \times H \times W] = [D, C \times H \times W]$ , where  $D$  is the output size. The affine transformation in the forward pass eq. (2.19) is then  $[D, C \times H \times W] \times [C \times H \times W, 1] = [D, 1]$ . The affine transformation in backward pass eq. (2.20) is the error having shape  $[1, 1, D, 1] = [D, 1]$  multiplied according to  $[D, C \times H \times W]^T \times [D, 1] = [C \times H \times W, 1] = [1, 1, C \times H \times W, 1] = [1, C, H, W]$ . Given the error for each neuron in a layer the filters and the bias gradients are found using eq. (2.21) and eq. (2.2) in their matrix multiplication forms.

### 3.1.5 The computational graph

As suggested by [12], using template computations will reduce implementation overhead when defining different neural network architectures. Therefore, a computational graph with template nodes is used.

The computations required by a fully connected layer, a convolutional layer, a max-pooling layer, the activation function, the dropout regularization and the loss function are all thought of as nodes in a graph. A node in the graph takes input of a specified shape and returns an output shape according to the nodes template parameter. Therefore, the only requirement for connecting these nodes, resulting in a computation according to eq. (1.1), is to make the output shape of each node

compatible with the input shape of the next node. In order to support any parameter setting of a layer the nodes are specified according to a template describing its parameters and input shape. The output shape for any layer can be deduced from these, as in the following figure



**Figure 3.2:** Example of a computational graph with template nodes. The input to the graph are 10 rgb images of size  $32 \times 32$ . These are convolved with 32 rgb filters of size  $5 \times 5$  using vertical and horizontal stride equal to 1 and padding equal to 2. The output is therefore, using eq. (2.23), a  $[10, 32, 32, 32]$  tensor. The activation layer activates the input which leaves the input shape unchanged. The maxpooling layer uses pooling 2 and stride 2 why the output width and height are halved. Finally, the fully connected layer converts the maxpooling output into an array of length  $32 \times 16 \times 16 = 8192$  which is multiplied with the weight matrix into a vector of length 5, where the 5 is chosen as example of a network that should make a prediction of the input having 5 possible class labels.

The computations performed in the above example can be extended to handle any input shape and parameter setting. Once this is implemented each node is reused many times and the only limitation is making the input shape and output shape compatible. For nodes which does not alter the input, this can be inferred from the previous layer output.

### 3.1.6 Context object pattern

The context object pattern is useful for propagating context information between different layers [13]. Each template node has an input edge and an output edge, as seen by figure (3.2). The input to the convolution layer is stored at the input edge. The context object takes the input at this edge, the parameters and a backend specification (CPU/GPU) and thereafter performs the computation as specified by the template node and puts the result at the output edge to this node. The next node thereafter has this output available by connecting its input edge to the previous

nodes output edge. The data is therefore propagated through the whole computational graph and put at the last nodes output, where it is given to the loss function. The same procedure goes for the backward propagation but now the context object also holds gradient data. The gradients are backpropagated from the output edge to the input edge of the last node, making them available for the previous node to backpropagate.

The separation of the computational specification with the computations performed allows for a variety within an implemented template computation. This variety could be using CPU instead of GPU or preferring another matrix multiplication library etc.

## 3.2 Efficient matrix multiplications

The most expensive computations in the computational graph stems from matrix multiplications of the affine transformations, as required by the fully connected layers and the convolution layers. There are many libraries which aims at optimizing the computational speed of multiplying matrices for CPU and GPU calculations. Matrix multiplications for both CPU and GPU are used (CPU in the case where no GPU was available) where the library for CPU computations is OpenBLAS and the library for GPU computations is cuDNN together with cuBLAS.

In practical applications a gain in matrix-matrix multiplications can be obtained by computing multiple inputs at once. This is known as stacking input images into a batch and yields increases performance over stochastic updating which uses only a single input image. The result on the gradients is that it is instead averaged over the batch such that the average steepest descend is taken. Due to the non-convex parameters space, estimating the gradient over the entire dataset, i.e. batch size equals to the full dataset, will not yield the globally optimal steepest descent and it is therefore suggested to use a smaller batch size to increase optimization speeds. The batch size, also called minibatch size when it is smaller than the full dataset size, is regarded as a hyperparameter which can be optimized.

### 3.2.1 OpenBLAS

OpenBLAS is an open source implementation of Basic linear algebra subprograms (BLAS) which implements common linear algebra operations as low-level routines optimized for the specific CPU at hand. The subroutine sgemm performs the matrix-matrix operation  $C = \alpha AB + \beta C$  with  $A$  and  $B$  being row-major ordered n-dimensional arrays, which with  $\alpha = 1$  and  $\beta = 0$  gives the desired matrix multiplication. The result is a n-dimensional array in row-major ordering.

### 3.2.2 cuDNN and cuBLAS

The computations on the GPU requires a NVIDIA supported graphics card together with CUDA software libraries [14]. The graphics card used is shown below

Manufacturer	PNY Technologies Inc.
Modell	Quadron K4200
Cuda cores	1344
Memory	4 GB
Memory bandwidth	173.0 GB/s

**Table 3.1:** *Graphics card used throughout this thesis.*

Among the software development kits is the Deep neural network library (cuDNN) and cuBLAS for GPU-accelerated computations. These computations uses parallelism on the graphic processing unit in order to speed up the affine transformation calculations.

cuDNN provides the convolution operations, maxpooling operations and activation operations for tensors stored as n-dimensional arrays in row-major ordering (NCHW). These operations are both the forward propagation and backward propagation through a layer. The fully connected operations are not available in the cuDNN software kit, but can be implemented using cuBLAS routines. The same matrix-matrix operations as in BLAS is available on the GPU, now called cublasSgemm. The matrix multiplication is again defined for two n-dimensional arrays in row-major ordering as;  $C = \alpha AB + \beta C$ . Using  $\alpha = 1$  and  $\beta = 0$  results in the desired matrix multiplication, but performed in parallel on the GPU.

### 3.3 Verify computations through gradient check

Given a correct implementation of the forward pass, any parameter gradient can be verified by the following procedure. Assume the problem of minimizing  $L(\theta)$  as a function of  $\theta$ . Suppose  $L : R \rightarrow R$  so  $\theta \in R$ . Assume the following gradient descent update rule

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta} \quad (3.3)$$

Now if there is a function  $g(\theta)$  which numerically computes  $\frac{\partial L}{\partial \theta}$  such that

$$\theta \leftarrow \theta - \eta g(\theta) \quad (3.4)$$

then  $g(\theta)$  can be approximated using

$$\frac{\partial}{\partial \theta} L(\theta) = \lim_{\epsilon \rightarrow 0} \frac{L(\theta + \epsilon) - L(\theta - \epsilon)}{2\epsilon} \quad (3.5)$$

with a sufficiently small  $\Delta$

$$g(\theta) \simeq \frac{L(\theta + \Delta) - L(\theta - \Delta)}{2\Delta} \quad (3.6)$$

Any implemented backward propagation can therefore be verified to be correct by comparing the parameter gradients obtained from using the implemented backpropagation functions to the parameter gradients obtained when approximating every

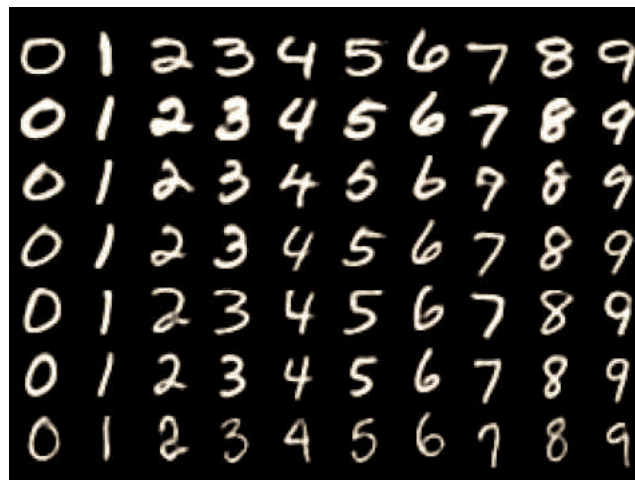
individual gradient by perturbing the respective parameter by  $\pm\Delta$ . Given a parameter gradient from the backpropagation functions and an approximated gradient these are compared using a relative error

$$\frac{|f'_a - f'_n|}{|f'_a + f'_n|} \quad (3.7)$$

where  $f'_a$  represents the approximated parameter gradient and  $f'_n$  represents the numerically computed parameter gradient. Summing over the relative error of each parameter gradients the result should be small, order of  $\sim 10^{-7} - 10^{-9}$ , in order for the computation to be numerically stable.

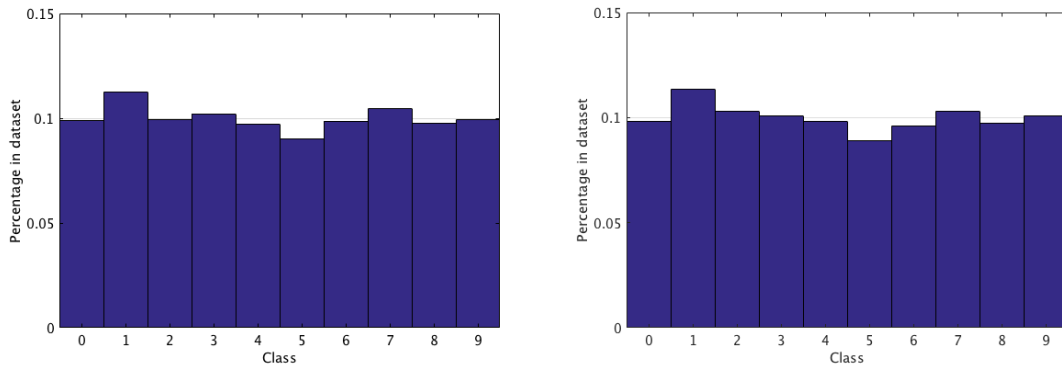
### 3.4 Implementation verification using the MNIST dataset

The MNIST database is a set of gray scale images of handwritten digits from 0 up to 9. The database contains a total of 60000 labeled training examples and 10000 test images.



**Figure 3.3:** *A few examples from the MNIST dataset.*

Each example is a  $28 \times 28$  image of black and white pixels, where white is foreground of value 0 and black is background of value 255. Each image contains a digit, which is centered using the center of mass of the pixels. The data in the set is stratified with approximately equally many occurrences of each class as seen by the following two dataset histograms.

(a) *Training set*(b) *Test set*

**Figure 3.4:** Figure (a) shows the histogram over the classes in the training set. Figure (b) shows the histogram over the classes in the test set.

### 3.4.1 Pre-processing

The Gaussian initialization assumes a zero centered symmetric distribution so each pixel value is transformed between  $-1$  and  $1$ . The mean value and standard deviation, computed over all training examples, is subtracted respectively divided by for each image. During testing, the same transforms are applied to each test example.

### 3.4.2 Network architecture

The architecture used is inspired from the LeNet-5 architecture [15] used for recognizing handwritten characters, but is modified by using only convolution, maxpooling and fully connected layers. The number of filters and their sizes are also modified by using only  $5 \times 5$  filtering and increasing the number of filters from  $6 \rightarrow 32$  in the first convolution layer and  $16 \rightarrow 64$  in the second convolution layer. For regularization purposes a dropout layer is added between the two fully connected layers. The architecture follows below.

Input size	Layer type	Parameters
$1 \times 28 \times 28$	Convolution	$5 \times 5$ , 32 filters, stride 1, padding 2
$32 \times 28 \times 28$	ReLU	
$32 \times 28 \times 28$	Maxpooling	pooling size = 2, stride = 2, padding = 0
$32 \times 14 \times 14$	Convolution	$5 \times 5$ , 64 filters, stride = 1, padding = 2
$64 \times 14 \times 14$	ReLU	
$64 \times 14 \times 14$	Maxpooling	pooling size = 2, stride = 2, padding = 0
3136	Fully connected	Output = 1024
1024	ReLU	
1024	Dropout	$p = 0.5$
1024	Fully connected	Output = 10
10	Softmax	

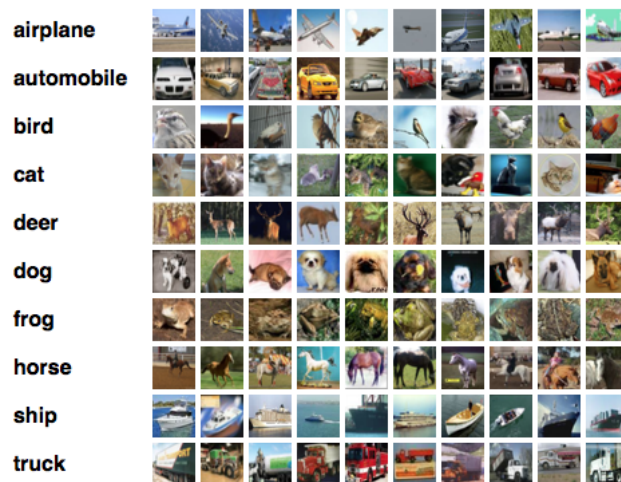
**Table 3.2:** Architecture used for MNIST classification.

### 3.4.3 Hyperparameters

The hyperparameters: learning rate, regularization strength, minibatch size and dropout rate were obtained through trial and error. The initial guessed learning rate, regularization strength, minibatch size and dropout rate were obtained from empirical estimations from previous reported results. The decay rates were initialized according to the authors (Kingma and Ba) suggestions.

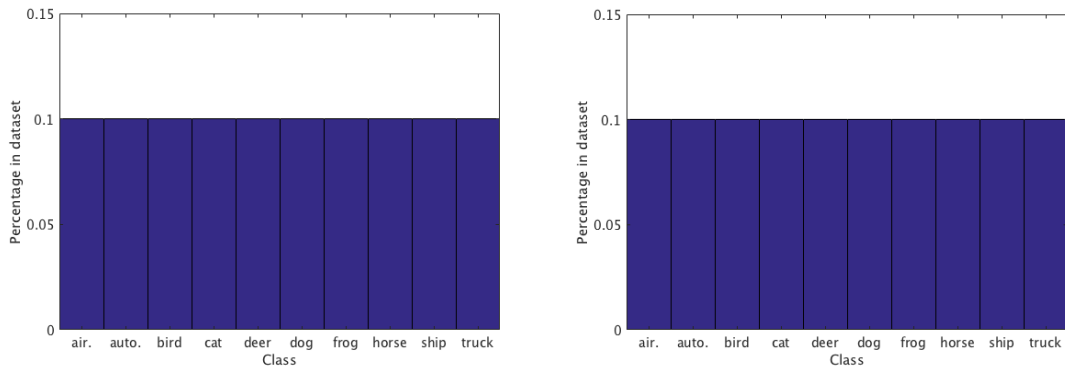
## 3.5 Challenging the CIFAR10 dataset

The CIFAR10 dataset consists of 60000 color images of size  $32 \times 32$  spread across 10 classes. The dataset is split into 50000 training images and 10000 test images.



**Figure 3.5:** *A few examples from the CIFAR10 dataset.*

The classes are mutually exclusive such that there is no overlap between the classes. This might seem like a great simplification however the score function can be modified such that the neural networks can rank the output probabilities. Thus, being able to accurately classify a mutual exclusive image can be generalized into more complex scenarios. The data in the set is stratified with equally many number from each class.

(a) *Training set*(b) *Test set*

**Figure 3.6:** Figure (a) shows the histogram over the classes in the training set. Figure (b) shows the histogram over the classes in the test set.

### 3.5.1 Pre-processing

The pixel values of each color channel R, G and B are between 0 and 255. These are scaled between  $-1$  and  $1$  and the mean image obtained from the training set is subtracted for each training and testing example.

### 3.5.2 Network architectures

Three architectures are used where in an attempt to gain high accuracy. The first one, Model A, is similar to the MNIST architecture except that it now takes RGB input. The second architecture, Model B, is an attempt to extend this model using more filter maps and an additional fully connected layer. The final model, Model C, is an extension of model B by using an additional convolution layer before each maxpooling layer.

Input size	Layer type	Parameters
$3 \times 32 \times 32$	Convolution	$5 \times 5$ , 32 filters, stride 1, padding 2
$32 \times 32 \times 32$	ReLU	
$32 \times 32 \times 32$	Maxpooling	pooling size = 2, stride = 2, padding = 0
$32 \times 16 \times 16$	Convolution	$5 \times 5$ , 64 filters, stride = 1, padding = 2
$64 \times 16 \times 16$	ReLU	
$64 \times 16 \times 16$	Maxpooling	pooling size = 2, stride = 2, padding = 0
4096	Fully connected	Output = 1024
1024	ReLU	
1024	Dropout	$p = 0.5$
1024	Fully connected	Output = 10
10	Softmax	

**Table 3.3:** Model A used for CIFAR10 classification. This model required 670MiB of GPU memory to run.



Input size	Layer type	Parameters
$3 \times 32 \times 32$	Convolution	$5 \times 5$ , 96 filters, stride 1, padding 2
$96 \times 32 \times 32$	ReLU	
$96 \times 32 \times 32$	Maxpooling	pooling size = 2, stride = 2, padding = 0
$96 \times 16 \times 16$	Convolution	$5 \times 5$ , 192 filters, stride = 1, padding = 2
$192 \times 16 \times 16$	ReLU	
$192 \times 16 \times 16$	Maxpooling	pooling size = 2, stride = 2, padding = 0
12288	Fully connected	Output = 2048
2048	ReLU	
2048	Dropout	p = 0.5
2048	Fully connected	Output = 2048
2048	ReLU	
2048	Dropout	p = 0.5
2048	Fully connected	Output = 10
10	Softmax	

**Table 3.4:** Model B used for CIFAR10 classification. This model required 2475MiB of GPU memory to run.

Input size	Layer type	Parameters
$3 \times 32 \times 32$	Convolution	$5 \times 5$ , 96 filters, stride 1, padding 2
$96 \times 32 \times 32$	ReLU	
$96 \times 32 \times 32$	Convolution	$5 \times 5$ , 96 filters, stride 1, padding 2
$96 \times 32 \times 32$	ReLU	
$96 \times 32 \times 32$	Maxpooling	pooling size = 2, stride = 2, padding = 0
$96 \times 16 \times 16$	Convolution	$5 \times 5$ , 192 filters, stride 1, padding 2
$192 \times 16 \times 16$	ReLU	
$192 \times 16 \times 16$	Convolution	$5 \times 5$ , 192 filters, stride 1, padding 2
$192 \times 16 \times 16$	ReLU	
$192 \times 16 \times 16$	Maxpooling	pooling size = 2, stride = 2, padding = 0
$8 * 8 * 192$	Fully connected	Output = 2048
2048	ReLU	
2048	Dropout	p = 0.5
2048	Fully connected	Output = 2048
2048	ReLU	
2048	Dropout	p = 0.5
2048	Fully connected	Output = 10
10	Softmax	

**Table 3.5:** Model C used for CIFAR10 classification. This model required 2493MiB of GPU memory to run.

### 3.5.3 Hyperparameters

The hyperparameters: learning rate, regularization strength, mini-batch size and dropout rate were obtained through trial and error (two trials attempt and how to see when the hyperparameters are incorrectly chosen are display in the appendix B). The initial learning rate, regularization strength, minibatch size and dropout rate were similar to that of the successful MNIST trials. The decay rates were initialized according to the authors (Kingma and Ba) suggestions.

## 3.6 Neural network architecture evaluations

Evaluation of each individual model is made using four different measurements.

### 3.6.1 Loss

The loss is evaluation of the negative log likelihood after each parameter update. This evaluation continues throughout every parameter update until the loss has stabilized around some value. This commonly happens after a few iterations through the entire dataset, where each iteration is referred to as an epoch. For clarity only each fiftieth loss evaluation is shown.

Due to the different number of parameters in the three CIFAR10 models, a novel normalization scheme is implemented to make the models loss curves comparable. The scheme is simply dividing each data point by the first loss value.

### 3.6.2 Accuracy

The accuracy is measured using a set of validation or test data and comparing the predicted and actual class label. The result is divided by the set size to obtain a probabilistic representation of the accuracy. Again for clarity, only the fiftieth accuracy evaluation is shown.

### 3.6.3 Confusion matrix

A useful way to evaluate the model is to check which classes are accurately predicted and which are not. This gives an estimation of the performance of the model when predicting a certain class among all classes, which may hindsight a possible model improvements. The measurement is performed by constructing a  $n \times n$  matrix  $M$  where  $n$  is the number of classes. Given a predicted class label  $\hat{y}$  and the actual label  $y_i$ , a one is added to the matrix at  $M(\hat{y}, y_i)$ . This is done for each example in the test set. To get a probability, each row is divided by its total amount. The ideal case is therefore to have a one at the diagonal and zero at each off diagonal element.

### 3.6.4 Weight visualization

One way to understand what the neural network has learned is to visualize the weights. There are techniques for how to visualize each weight in a network, how-

ever in this work only the filters in the first convolutional layer are displayed. These filters are applied directly onto the input image and are therefore easy to interpret.

The weights are both regularized and operates onto pre-processed data, why the coloring scheme is based on representing relative differences in weight values for each filter. For each filter, this set of weights  $w$  are normalized by

$$w = \frac{w - \min(w)}{\max(w) - \min(w)} \quad (3.8)$$

thereafter the appropriate scaling is applied. For grayscale each value is scaled within the interval  $[0, 255]$ , where 0 is represented by a black pixel and a 1 represents a white pixel. The rgb images are made by first transforming the values to the lie on the interval  $[0, 1]$  and thereafter concatenate each red, green and blue channel. The following table shows which color is obtained for a few combinations of color values.

Red	Green	Blue	Color
1	1	1	white
0	0	0	black
1	0	0	red
0	1	0	green
0	0	1	blue
1	1	0	yellow
1	0	1	pink
0	1	1	light blue
0.5	0.5	0.5	gray



# 4

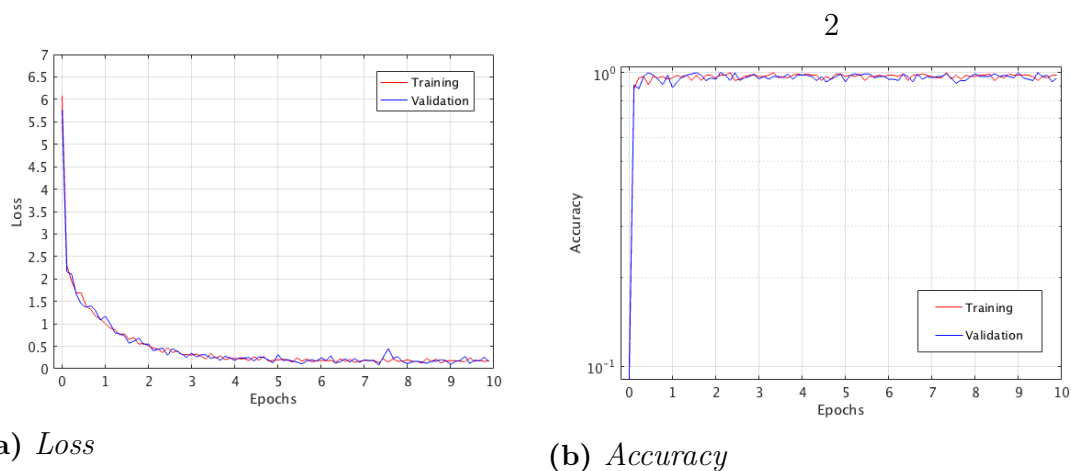
## Results

### 4.1 MNIST classification

The training (optimization) consists of using the entire dataset but splitting 75% into a training set and the rest into a validation set. For each training batch a corresponding validation batch is used to estimate the ability for the neural network to generalization to unseen data. The validation batches are chosen with a random starting point in the validation set. Thereafter, the number required by the batch size is selected as each example following this randomly chosen one. The reason is to maintain stratified data of approximately equal many classes.

Hyperparameter	Value
Learning rate ( $\eta$ )	0.0001
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Minibatch size ( $B$ )	100
Dropout rate ( $p$ )	0.75

**Table 4.1:** *Hyperparameters used for MNIST classification.*



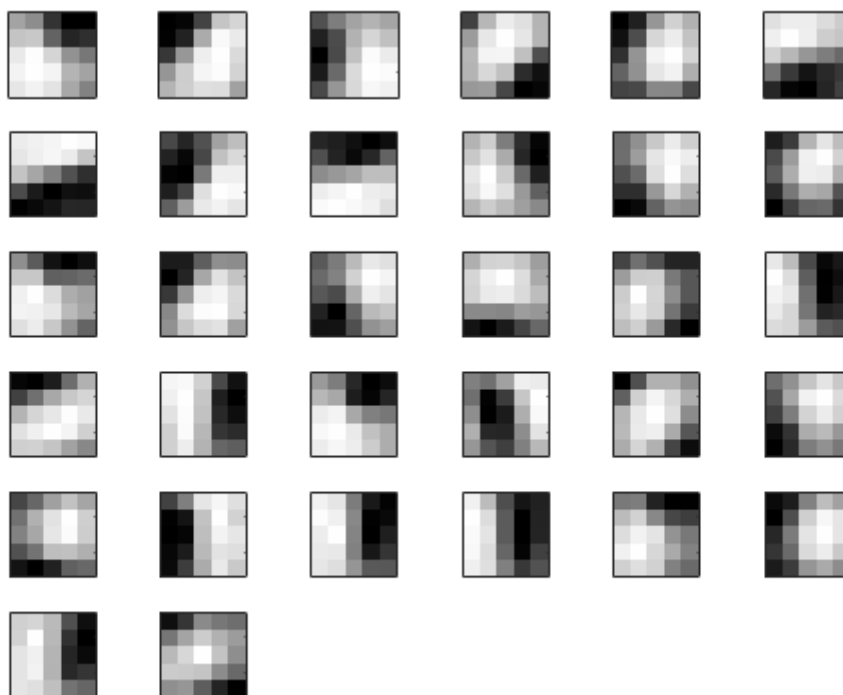
**Figure 4.1:** *Result from testing on the MNIST dataset. Figure (a) shows the loss. Figure (b) shows the accuracy.*

## 4. Results

---

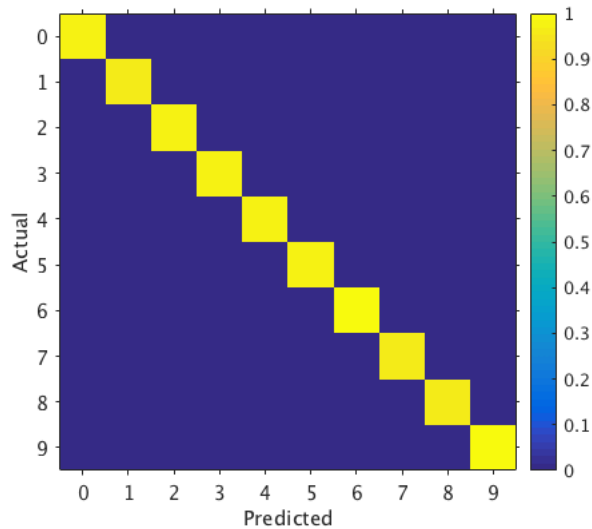
Running the trained neural network architecture on the test data yields an accuracy of 97.31%.

The 32 filters used by the first convolution layer are applied directly onto the input image. This gives a good intuition about what the neural network tries to use for classification and are shown in figure (4.2).



**Figure 4.2:** *The first 32 filters which the neural network learned to use when classifying handwritten digits.*

The confusion matrix over the predicted and actual classes are shown in figure (4.3).



**Figure 4.3:** *MNIST confusion matrix.*

### 4.1.1 Discussion

Deep neural networks has shown to handle the MNIST dataset with ease and it is by todays standard viewed as a toy example. Similar results, and better ones, have been reported on the dataset using similar neural network architectures, so the accuracy was expected. However, the dataset is useful in verifying that multiple implementation details such as the weight initialization and the gradient check are reasonable for the network architecture at hand. For the architecture used, a numerical error of the order  $\sim 10^{-9}$  on both the CPU and GPU were observed when using  $\Delta = 10^{-4}$ .

Moreover, the learned filters by the first convolution layer contains many different kinds of horizontal, vertical and diagonal edges which makes intuitive sense since different types of edge detectors seems like a useful measurement in classifying the digits. But, there also seems to exists multiple occurrences of the same filter and whether or not they are useful needs further investigation. It could be the case that there are more filters then necessary.

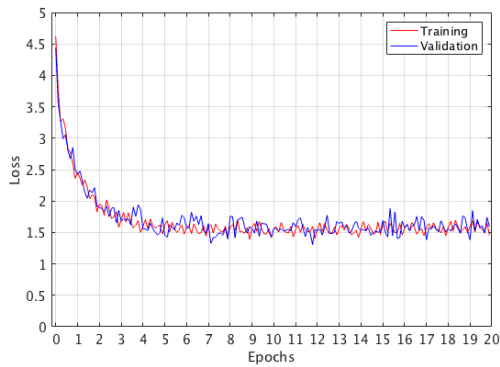
## 4.2 CIFAR10 classification

The training is performed using the whole dataset and splitting 90% into a training set and the rest into a validation set. Same validation batch strategy is used as this will keep the data stratified with approximately equally many classes in each batch.

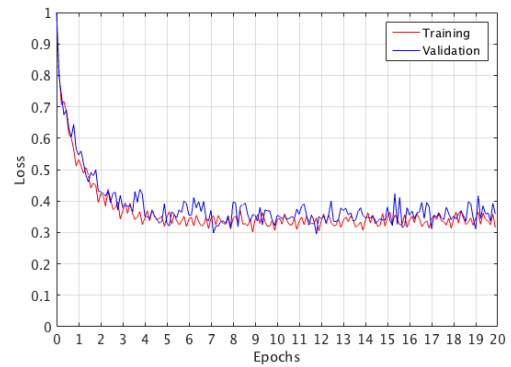
## 4.2.1 Model A

Hyperparameter	Value
Learning rate ( $\eta$ )	0.0005
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Minibatch size ( $B$ )	100
Dropout rate ( $p$ )	0.5

Table 4.2: Hyperparameters used for CIFAR10 classification using model A.



(a) Loss



(b) Normalize loss

Figure 4.4: Result from training model A. Figure (a) shows the unnormalized loss. Figure (b) shows the normalized loss.

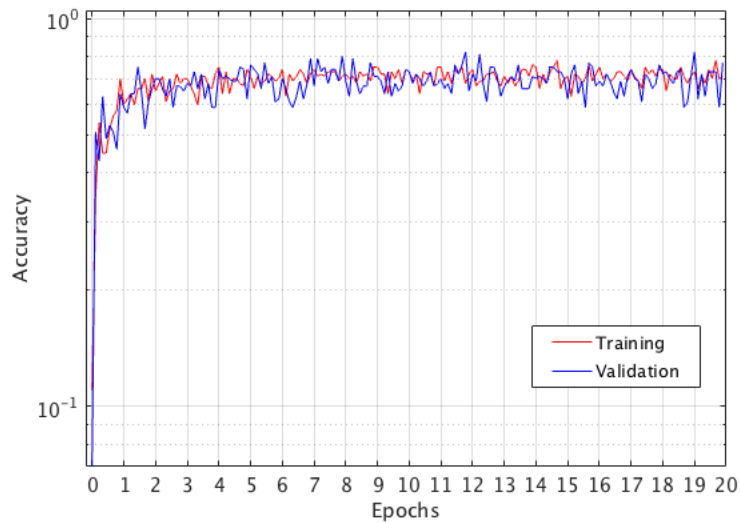
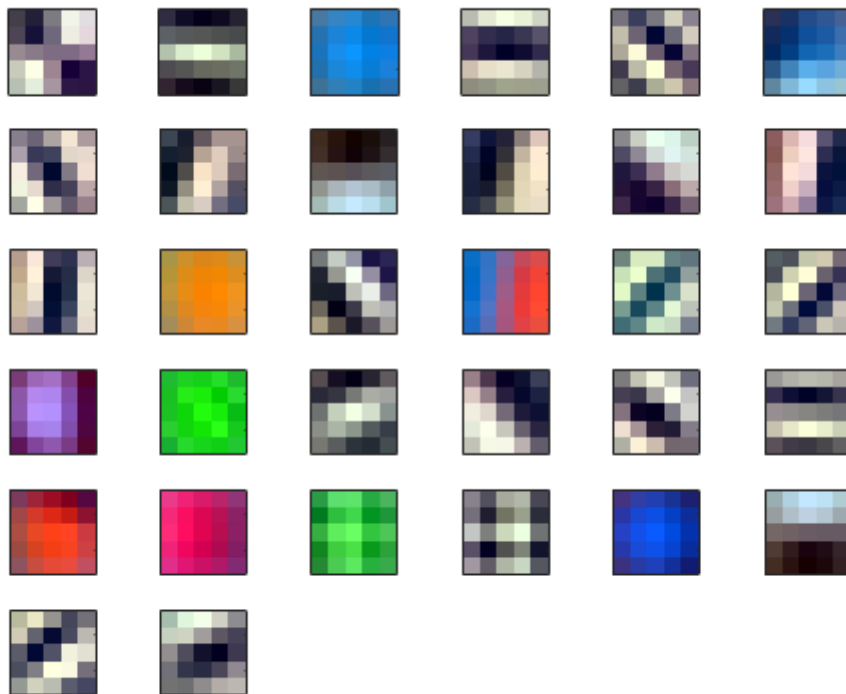


Figure 4.5: The accuracy during training.



The trained neural network architecture obtains an accuracy of 70.39% when run on the test data.

The 32 filters used by the first convolution layer is shown in figure (4.6).



**Figure 4.6:** *The first 32 filters which the neural network learned from classifying the CIFAR10 classes.*

The confusion matrix over the predicted and actual classes are shown in figure (4.7).

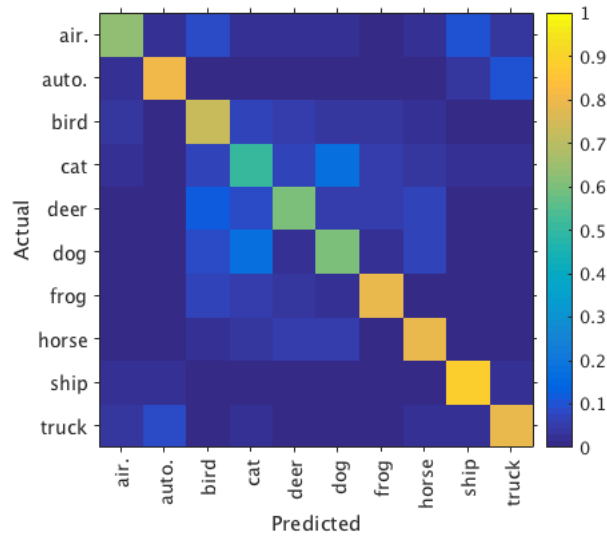
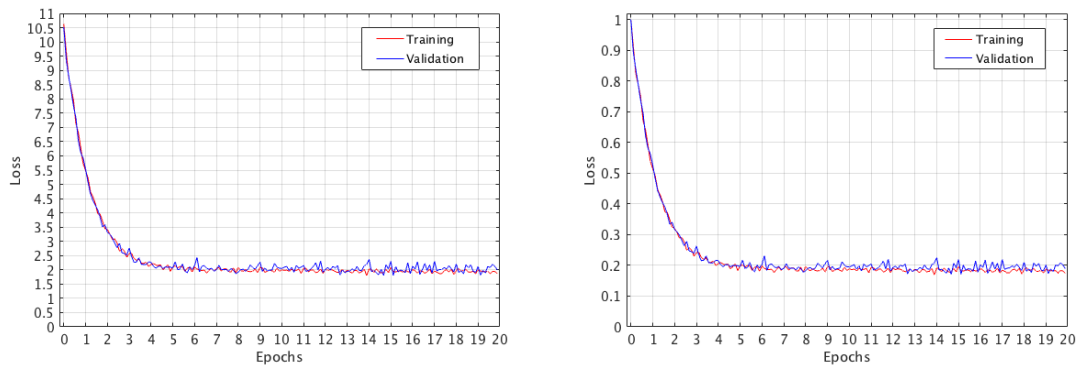


Figure 4.7: Model A confusion matrix.

### 4.2.2 Model B

Hyperparameter	Value
Learning rate ( $\eta$ )	0.0005
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Minibatch size (B)	100
Dropout rate (p)	0.5

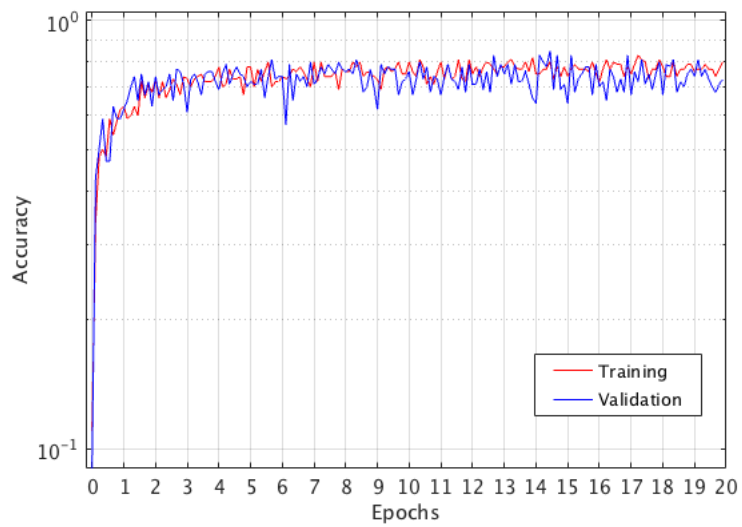
Table 4.3: Hyperparameters used for CIFAR10 classification using model B.



(a) Loss

(b) Loss normalized

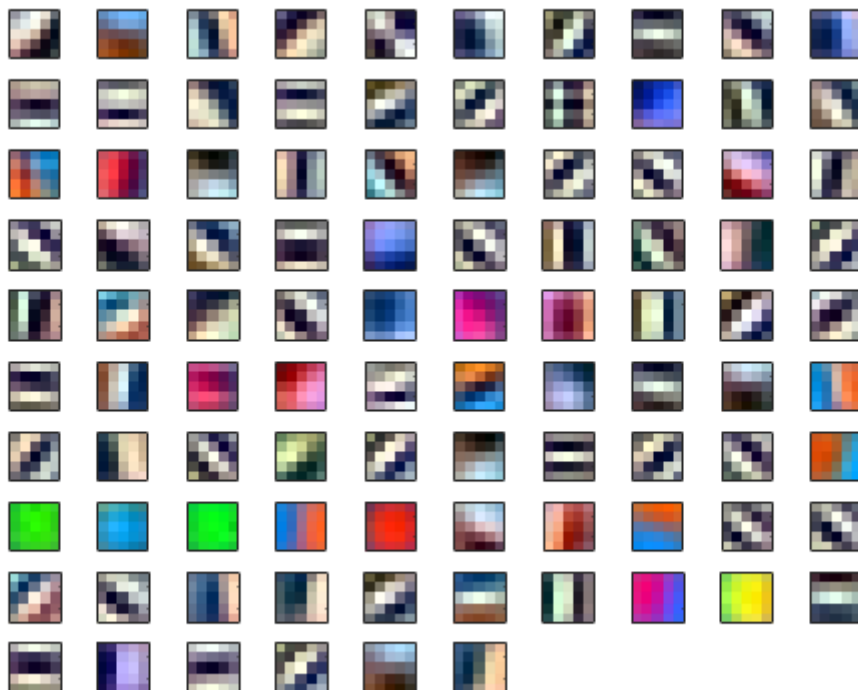
Figure 4.8: Result from training model B. Figure (a) shows the unnormalized loss. Figure (b) shows the normalized loss.



**Figure 4.9:** *The accuracy during training.*

The trained neural network architecture obtains an accuracy of 72.64% when run on the test data.

The 96 filters used by the first convolution layer is shown in figure (4.10).



**Figure 4.10:** *The first 96 filters which model B learned from classifying the CI-FAR10 classes.*

The confusion matrix over the predicted and actual classes are shown in figure (4.11).

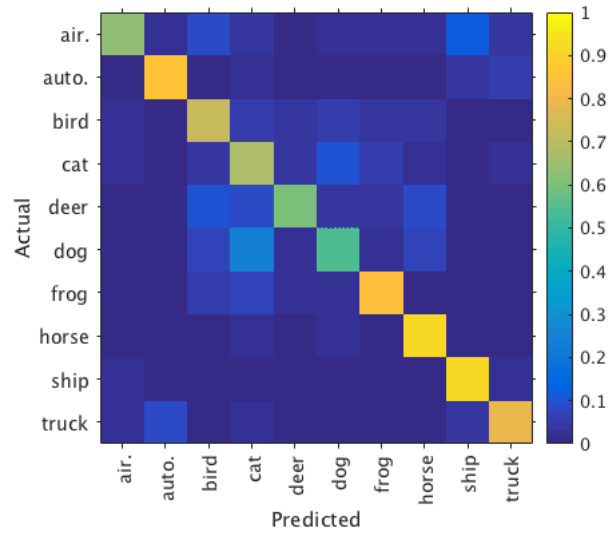
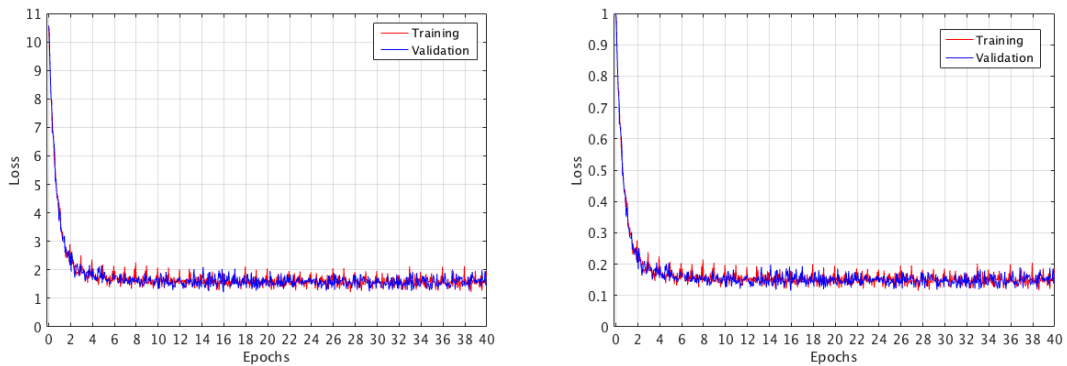


Figure 4.11: Model B confusion matrix.

### 4.2.3 Model C

Hyperparameter	Value
Learning rate ( $\eta$ )	0.0005
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Minibatch size (B)	50
Dropout rate (p)	0.5

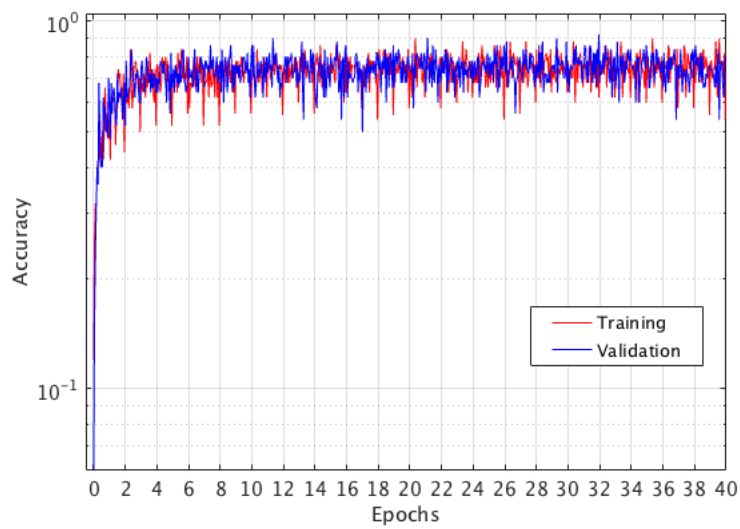
Table 4.4: Hyperparameters used for CIFAR10 classification using model C.



(a) Loss.

(b) Normalized loss.

Figure 4.12: Result from training model C. Figure (a) shows the unnormalized loss. Figure (b) shows the normalized loss.



**Figure 4.13:** *The accuracy during training.*

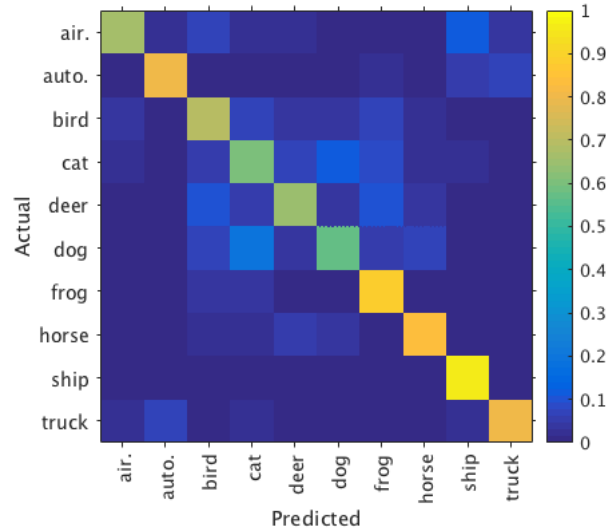
The trained neural network architecture obtains an accuracy of 72.88% when run on the test data.

The 96 filters used by the first convolution layer is shown in figure (4.14).



**Figure 4.14:** *The first 96 filters which model C learned from classifying the CI-FAR10 classes.*

The confusion matrix over the predicted and actual classes are shown in figure (4.15).



**Figure 4.15:** *Model C confusion matrix.*

## 4.2.4 Discussion

### 4.2.4.1 Model A

During the first 5 epoch the loss drops from about 4.5 to 1.5 where after it remains stable through the rest of the epochs. Similarly, the accuracy stabilizes around 0.7 for the both training and validation set. There is no sign of overfitting yet, which indicates a good combination between dropout and regularization strength but can be a result of the small validation set.

Similarly to the MNIST filters there exists clear structures in the model A filters. Both horizontal (e.g. second and fourth in the first row), vertical (first in the third row) and diagonal types of edge detection are found among the filters. Despite the introduction of red, green and blue colors the edge detectors remains gray meaning they value edges independently of color. The color variety also introduces filters of different periodicity. The second last filter in the third row displays this characteristic. This periodicity will be returned to in the next model discussion. A sign of fitting to local structure exists in the filters displaying a single to two colors. These filters would not exist in a gray scaling setting of the images.

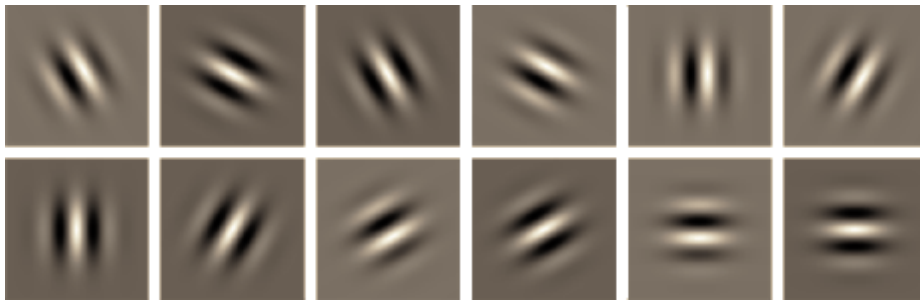
The confusion matrix shows clearly which classes were difficult to classify. Dogs and deers ended up being classified no better than fifty-fifty. Interestingly, the airplanes showed some difficulty to classify. The overall result of  $\sim 70\%$  reflects the difficulty in classification among the dog and deer classes.

#### 4.2.4.2 Model B

The loss drops from about 10.5 to 2 in the first 5 epochs where after it remains stable. A slight sign of overfitting is shown at the end where the training loss remains on average more often below the validation loss. The accuracy stabilizes around 0.75 where it remains throughout the training.

The filters display, similarly to the previous model, vertical, horizontal and diagonal edges. In general there are simply more of each filter compared to the previous model. But, with the additional filters the model seems to prefer to measure periodicity. Again there seems to be fitting to local structures by having filters of one to two colors. A slight improvement of 2.25% was obtained over the previous model by mainly increasing the correct classification rate of the cat class. A claim why the neural networks tries to learn various periodicity's and why this improves the accuracy is made.

Claim; The periodicity's has a correspondence with Gabor filters, which are models of the 'simple' cells in the visual cortex of mammalian brains [16, 17]. Thus, analyzing images using Gabor filters is thought to be similar to perception in the human visual system.



**Figure 4.16:** *Gabor filters of various orientations.*

From the assembling of the dataset it is easy to conclude that the human visual system is very efficient at distinguishing the classes in the natural images and so it makes sense that the neural network aims at utilizing similar distinguishing techniques in its classification scheme. However, whether this correspondence claim is true or not remains to be more thoroughly investigated, through e.g. pre-processing the images by applying various forms of Gabor filters and see how this affects the final accuracy. Additionally, the final model did not exhibit any similar structure in its learned weights and whether Gabor filter is the most optimal strategy or most generalizable strategy also remains to further investigate.

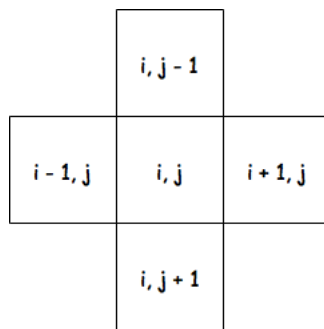
#### 4.2.4.3 Model C

The loss drops from about 10.5 to 2 in the first 2 epochs where after it slowly converges to around 1.5. Due to the slower convergence the training was run for 40

epochs rather than 20. Due to reduced minibatch size there is an increase in amplitude of the deviation from the average. The accuracy stabilizes around 0.7 with a slight sign of overfitting.

The filters however have changed and there are no more periodicities. When using two convolutional layers before the maxpooling layer there seems to be a tendency to instead learn pixelwise neighborhoods rather than edges or set of pixel structures. There are multiple diagonal edge detectors but they seem to correspond to local structure since they are color dependent. A small improvement of 0.24 was obtained over the previous model. It is difficult to attribute this to any specific improvement among the classes.

Performing an exhaustive search by binarizing each region via correspondence of individual pixel neighborhoods is naturally a valid approach and it seems as if this neural network model tries to perform something similar. The filters without colors having a black or white pixel in the center, seems to be using mainly a von Neumann neighborhoods of Manhattan distance  $r = 1$ .



**Figure 4.17:** *von Neumann neighborhood of Manhattan distance  $r = 1$ .*

When using a stride equal to 1 in both the vertical and horizontal direction each affine transformation will correspond to a pixelwise neighborhood summation. Using black, gray and white pixel neighborhoods the first convolutional layer can check whether a pixel at a location, within the neighborhood around a center in the input, is absent or not and if so, by what amount.

There are two distinct filter types of this characteristics. The first type contains a black pixel at the center and various gray levels at the four neighborhood locations. A few of these types have off centered black pixels but despite this the neighborhood has the same four neighbors. The affine transformation used by this filter type measures the surrounding pixels at each location around the current center since the center term is canceled. The neighbors are weighted differently which indicates an attempt to break the neighborhood symmetry between these filter types. The second type completely cancels the surrounding pixels and keeps only a small subset. The affine transformation used by this filter type therefore measures only within the von Neumann neighborhood. Again, symmetry between the filters is broken by using only three out of four neighborhood locations. Both filter types also exists with



some colors but colors are non-generic properties and it is difficult to evaluate how these related to the specific data. By extracting the distribution of colors images of the same class should reveal if a bias infers color dependencies. It should also be noted that the von Neumann neighborhood with  $r = 1$  only requires 9 pixels for the two types of filters discussed above. Therefore, it could be useful to reduce the filter size to  $3 \times 3$  instead of  $5 \times 5$ , however attempts with this filter sizes was not successful (in terms of accuracy which was  $\sim 50\%$  and with high amount of overfitting) and needs further experimentation.

Furthermore, it is difficult to say how this information is utilized by later layers in the neural network using the current visualization scheme. Thus, an extension of the visualization scheme is required in order to gain more insight into what the neural network has learned.

#### 4.2.4.4 Comparison with the state-of-the-art

The current state-of-the-art on the CIFAR10 dataset is the Fractional-Maxpooling neural network [18] with an accuracy of 96.53%, surpassing the estimated human level performance of  $\sim 94\%$ . This was obtain through using a twelve layer deep convolutional neural network where the number of filters increases linearly by  $160n$ , where  $n$  is the layer number. Due to memory limitations of the graphics card used, a model with similarly many filters and layers could not be attempted. The learned weights of the state-of-the-art layers was not shown either and such a comparison could not be performed. Nonetheless, there is a clear improvement to be made on the models to reach these accuracies, some improvements are discussed in the following sections.

#### 4.2.5 Pre-processing

The amount of pre-processing is a debatable topic since it may or may not affects the information content confined within the data. But, as indicated by the learned filters from the MNIST dataset, edge detection seems useful for the neural network to learn. With many data transforms aimed at highlighting edges and textures, pre-processing the images in this manner might lead to better performance on the dataset. If such a local performance boost also increases the general performance is difficult to answer and needs further experimentation across several datasets.

Another pre-processing technique, which is mainly aimed at reducing overfitting, is data augmentation. The idea is to randomly crop, rotate and mirror the data to generate new data from the existing set. This has show to reduce overfitting and also increase accuracy slightly. This technique should however be evaluated before use, since it would be rather risky to apply for a dataset like MNIST.

Finally, using RGB color space where the intensity is mixed into the R, G and B values could be reconsidered. Using for example the YCbCr color space where the intensity is separated from the colors could prove more useful as the image classification should be independent of the intensity level.

### 4.2.6 Optimizing the neural network architectures

Evaluating the neural network architectures is mainly up to the computational power available. The training of the model B took  $\sim 6$  hours and model C took  $\sim 60$  hours and so performing an optimization scheme over several neural network architectures was not a possibility under the current circumstances. With more computational power multiple neural networks could be run simultaneously where after the best performing once are iterated further upon etc. This evaluation scheme was however not possible to perform.

A scheme showing whether the computational power available was fully used would be to measure neuron activity. Deep neural networks contains millions of neurons and whether a single neuron is active during the entire training and testing process could indicate how the neural network architecture performs. With a high ratio of inactive neurons, GPU memory is unused and neural network structure could be reconsidered.

Another scheme which comes with a bit more computational power is to perform a hyperparameter search to optimize these, which has shown to give an improvements of a few percentages. However, with the large gap up to the current state-of-the-art due to more computational power and larger neural network architectures, trying to optimize the performance of the current models is not of highest priority.

Comparing model A, model B, model C and the current state-of-the-art there seems to show a general pattern of increased number of filters improves the performance. Whether this is true in general remains to investigate.

# 5

## Conclusions

Deep feedforward neural networks shows great capabilities in handwritten digit classification and a good potential in natural image classification. The accuracy aimed for was obtained for clear distinguishable classes however not on classes involving many similarities such as dogs, deers and cats. Higher accuracy has been reported before this work, but despite this the question in how to construct these neural networks still remains open. By trying different models I address the question of what is and can be learned by the neural network. The neural network does learn desired features known to be effective in general for natural image classification but does also try to optimize non-generic structures. Regarding the ability to generalize, the models used shows both the ability to learn general features, such as edges, but do largely learn non-generic strictures.

The common understanding and criticism of deep learning remains in the black-box perspective and by addressing the various components and their very specific role in the neural network together with showing what is learned, this thesis gives a better understanding of the potentials. If the neural network truly learns Gabor filters, which do have a correspondence with other biological components without given the explicit knowledge about those, then the field shows great potential in being applied in various natural image classification task. However, with the amount of time having to be invested into the technical details in their optimization, the amount of data required and having to close to 'guess' different neural network models there is still more elaborate work to be performed before they are 'algorithmic' enough to straight forward implement and use. Additionally, the thesis has given results which a theoretical model has to be able to exhibit under the same neural network model settings.

### 5.1 Future work

As for future work it would be interesting to study the role of Gabor filters in learning to classify natural images as they seem to contain important information. It would also be interesting to develop knowledge behind the choice of the neural network to use von Neumann neighborhood connections and what that would biologically correspond to, if such a correspondence exists. Which of the two choices is 'right' or better in general would also be interesting to further investigate.

With increased computational power, such that neural network architecture opti-

## 5. Conclusions

---

mization could be performed efficiently, many more models, filter sizes and other hyperparameters could be investigated. This could improve understanding in the development of deep feedforward neural network models and what is important to optimize, which might lead to new areas of applications.

# Bibliography

- [1] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Mazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves, and J. Welling. Never-ending learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, 2015.
- [2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [3] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.
- [4] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL <http://dx.doi.org/10.1007/BF02478259>.
- [5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL <http://dx.doi.org/10.1007/BF02551274>.
- [6] Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2015.
- [7] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL <http://arxiv.org/abs/1412.6980>.
- [8] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.
- [11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep

- into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL <http://arxiv.org/abs/1502.01852>.
- [12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012. URL <http://arxiv.org/abs/1206.5533>.
- [13] Arvind S Krishna, Douglas C Schmidt, and Michael Stal. Context object a design pattern for efficient information sharing across multiple system layers.
- [14] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL <http://doi.acm.org/10.1145/1365490.1365500>.
- [15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [16] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.
- [17] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.
- [18] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014. URL <http://arxiv.org/abs/1412.6071>.

# A

## Details to formulas in theory section

### A.1 Negative log-likelihood gradient

$$\frac{\partial L(y_i, \hat{\mathbf{y}})}{\partial \hat{y}_k} = \frac{\partial}{\partial \hat{y}_k} (-\log(p(\hat{\mathbf{y}})_{y_i})) \quad (\text{A.1})$$

$$= \frac{\partial}{\partial \hat{y}_k} \left( -\log \left( \frac{e^{\hat{y}_{y_i}}}{\sum_j e^{\hat{y}_j}} \right) \right) \quad (\text{A.2})$$

$$= \frac{\partial}{\partial \hat{y}_k} \left( -\hat{y}_{y_i} + \log \left( \sum_j e^{\hat{y}_j} \right) \right) \quad (\text{A.3})$$

$$= -\frac{\partial \hat{y}_{y_i}}{\partial \hat{y}_k} + \frac{\partial}{\partial \hat{y}_k} \log \left( \sum_j e^{\hat{y}_j} \right) \quad (\text{A.4})$$

$$= \begin{cases} p(\hat{\mathbf{y}})_k - 1 & \text{if } k = y_i \\ p(\hat{\mathbf{y}})_k & \text{otherwise} \end{cases} \quad (\text{A.5})$$

### A.2 Backpropagation: Fully connected layer

By definition

$$\delta_k^{l-1} = \frac{\partial L}{\partial z_k^{l-1}} \quad (\text{A.6})$$

giving

$$\delta_k^{l-1} = \frac{\partial L}{\partial z_k^{l-1}} \quad (\text{A.7})$$

$$= \sum_j \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial z_k^{l-1}} \quad (\text{A.8})$$

$$= \sum_j \frac{\partial z_j^l}{\partial z_k^{l-1}} \delta_j^l \quad (\text{A.9})$$

and

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad (\text{A.10})$$

$$= \sum_k w_{jk}^l f(z_k^{l-1}) + b_j^l \quad (\text{A.11})$$

the differentiation simplifies to

$$\frac{\partial z_j^l}{\partial z_k^{l-1}} = \frac{\partial}{\partial z_k^{l-1}} \left( \sum_{k'} w_{jk'}^l f(z_{k'}^{l-1}) + b_j^l \right) \quad (\text{A.12})$$

$$= w_{jk}^l f'(z_k^{l-1}) \quad (\text{A.13})$$

since only the term where  $k = k'$  is non-zero. Substituting into eq. A.7 yields

$$\delta_k^{l-1} = \sum_j w_{jk}^l \delta_j^l f'(z_k^{l-1}) \quad (\text{A.14})$$

Analogously,

$$\frac{\partial L}{\partial w_{jk}^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} \quad (\text{A.15})$$

$$= \frac{\partial z_j^l}{\partial w_{jk}^l} \delta_j^l \quad (\text{A.16})$$

where

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l} \left( \sum_{k'} w_{jk'}^l f(z_{k'}^{l-1}) + b_j^l \right) \quad (\text{A.17})$$

$$= f(z_k^{l-1}) \quad (\text{A.18})$$

since only the term where  $k = k'$  is non-zero. Substituting into eq. (A.10) yields

$$\frac{\partial L}{\partial w_{jk}^l} = f(z_k^{l-1}) \delta_j^l \quad (\text{A.19})$$

$$= a_k^{l-1} \delta_j^l \quad (\text{A.20})$$

Lastly,

$$\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \quad (\text{A.21})$$

$$= \delta_j^l \quad (\text{A.22})$$

since

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \quad (\text{A.23})$$



### A.3 Backpropagation: Convolutional layer

By definition

$$\delta_{jk}^{l-1} = \frac{\partial L}{\partial z_{jk}^{l-1}} \quad (\text{A.24})$$

giving

$$\delta_{jk}^{l-1} = \frac{\partial L}{\partial z_{jk}^{l-1}} \quad (\text{A.25})$$

$$= \sum_{j'} \sum_{k'} \frac{\partial L}{\partial z_{j'k'}^l} \frac{\partial z_{j'k'}^l}{\partial z_{jk}^{l-1}} \quad (\text{A.26})$$

$$= \sum_{j'} \sum_{k'} \frac{\partial z_{j'k'}^l}{\partial z_{jk}^{l-1}} \delta_{j'k'}^l \quad (\text{A.27})$$

and

$$z_{j'k'}^l = \sum_{j''} \sum_{k''} w_{j''k''}^l a_{j'-j'', k'-k''}^{l-1} + b_{j'k'}^l \quad (\text{A.28})$$

the differentiation becomes

$$\frac{\partial z_{j'k'}^l}{\partial z_{jk}^{l-1}} = \frac{\partial}{\partial z_{jk}^{l-1}} \left( \sum_{j''} \sum_{k''} w_{j''k''}^l a_{j'-j'', k'-k''}^{l-1} + b_{j'k'}^l \right) \quad (\text{A.29})$$

$$= \frac{\partial}{\partial z_{jk}^{l-1}} \left( \sum_{j''} \sum_{k''} w_{j''k''}^l f(z_{j'-j'', k'-k''}^{l-1}) + b_{j'k'}^l \right) \quad (\text{A.30})$$

$$= w_{j'-j, k'-k}^l f'(z_{j,k}^{l-1}) \quad (\text{A.31})$$

since only the terms where  $j = j' - j''$  and  $k = k' - k''$  are non-zero. Substituting into eq. A.25 yields

$$\delta_{jk}^{l-1} = \sum_{j'} \sum_{k'} w_{j'-j, k'-k}^l \delta_{j'k'}^l f'(z_{j,k}^{l-1}) \quad (\text{A.32})$$

$$= \delta_{jk}^l * w_{-j, -k}^l f'(z_{j,k}^{l-1}) \quad (\text{A.33})$$

$$= \delta_{jk}^l * \text{rot}_{180^\circ}(w_{jk}^l) f'(z_{j,k}^{l-1}) \quad (\text{A.34})$$

Analogously,

$$\frac{\partial L}{\partial w_{jk}^l} = \sum_{j'} \sum_{k'} \frac{\partial L}{\partial z_{j'k'}^l} \frac{\partial z_{j'k'}^l}{\partial w_{jk}^l} \quad (\text{A.35})$$

$$= \sum_{j'} \sum_{k'} \delta_{j'k'}^l \frac{\partial z_{j'k'}^l}{\partial w_{jk}^l} \quad (\text{A.36})$$

the differentiation becomes

$$\frac{\partial z_{j'k'}^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l} \left( \sum_{j''} \sum_{k''} w_{j''k''}^l a_{j'-j'', k'-k''}^l + b_{j'k'}^l \right) \quad (\text{A.37})$$

$$= \frac{\partial}{\partial w_{jk}^l} \left( \sum_{j''} \sum_{k''} w_{j''k''}^l f(z_{j'-j'', k'-k''}^{l-1}) + b_{j'k'}^l \right) \quad (\text{A.38})$$

$$= f(z_{j'-j, k'-k}^{l-1}) \quad (\text{A.39})$$

since only the terms where  $j = j''$  and  $k = k''$  are non-zero. Substituting into eq. A.35 yields

$$\frac{\partial L}{\partial w_{jk}^l} = \sum_{j'} \sum_{k'} \delta_{j'k'}^l f(z_{j'-j, k'-k}^{l-1}) \quad (\text{A.40})$$

$$= \delta_{jk}^l * f(z_{-j, -k}^{l-1}) \quad (\text{A.41})$$

$$= \delta_{jk}^l * f(\text{rot}_{180^\circ}(z_{jk}^{l-1})) \quad (\text{A.42})$$

Lastly,

$$\frac{\partial L}{\partial b_{jk}^l} = \frac{\partial L}{\partial b^l} \quad (\text{A.43})$$

$$= \sum_{j'} \sum_{k'} \frac{\partial L}{\partial z_{j'k'}^l} \frac{\partial z_{j'k'}^l}{\partial b^l} \quad (\text{A.44})$$

$$= \sum_{j'} \sum_{k'} \delta_{j'k'}^l \quad (\text{A.45})$$

since

$$\frac{\partial z_{j'k'}^l}{\partial b^l} = 1 \quad (\text{A.46})$$

## A.4 Gaussian weight initialization

Following the authors arguments [8], but considering any affine neuron with ReLU activation instead:

Consider an affine transformation

$$z^l = \sum_{i=1}^n w_i^l a_i^{l-1} = w_1^l a_1^{l-1} + w_2^l a_2^{l-1} + \dots + w_n^l a_n^{l-1} \quad (\text{A.47})$$

where  $w^l$  are the weights in layer  $l$  and  $a^{l-1}$  is the activated output from layer  $l-1$ . The same assumptions as the authors are made; the weights  $w^l$  and the input  $a^{l-1}$  are all mutually independent and share the same zero-centered distribution. The variance in the output is therefore related to the variance in the input by

$$\text{Var}[z^l] = \sum_{i=1}^n \text{Var}[w_i^l a_i^{l-1}] \quad (\text{A.48})$$

The variance in any term  $w_i^l a_i^{l-1}$  is

$$\text{Var}[w_i^l a_i^{l-1}] = \text{E}[(w_i^l a_i^{l-1})^2] - (\text{E}[w_i^l a_i^{l-1}])^2 \quad (\text{A.49})$$

$$= \text{E}[(w_i^l)^2] \text{E}[(a_i^{l-1})^2] - (\text{E}[w_i^l])^2 (\text{E}[a_i^{l-1}])^2 \quad (\text{A.50})$$

$$= \text{Var}[a_i^{l-1}] \text{E}[w_i^l]^2 + \text{Var}[w_i^l] \text{E}[a_i^{l-1}]^2 + \text{Var}[a_i^{l-1}] \text{Var}[w_i^l] \quad (\text{A.51})$$

$$= \text{Var}[w_i^l] \text{E}[(a_i^{l-1})^2] \quad (\text{A.52})$$

If  $w^{l-1}$  has a symmetric distribution around zero, then  $a^{l-1}$  has zero mean and is a symmetric distribution around zero. This leads to,

$$\text{E}[(a_i^{l-1})^2] = \frac{1}{2} \text{Var}[z^{l-1}] \quad (\text{A.53})$$

since ReLU activation halves a symmetric distribution with zero-mean. With eq. (A.49) this becomes

$$\text{Var}[w_i^l a_i^{l-1}] = \frac{1}{2} \text{Var}[w_i^l] \text{Var}[z^{l-1}] \quad (\text{A.54})$$

So with  $L$  layers put together,

$$\text{Var}[z^l] = \text{Var}[z^1] \left( \prod_{l=2}^L \frac{1}{2} n \text{Var}[w^l] \right) \quad (\text{A.55})$$

and maintaining the variance leads to the condition

$$\frac{1}{2} n \text{Var}[w^l] = 1 \quad \forall l \quad (\text{A.56})$$

Therefore,  $w^l$  should be initialized to a zero-mean Gaussian distribution whose standard deviation is

$$\sqrt{\frac{2}{n}} \quad (\text{A.57})$$

where  $n$  is the number of incoming weight connections to each output activation. For the first layer, which does not have an activation function applied onto the input, the Xavier initialization is valid.



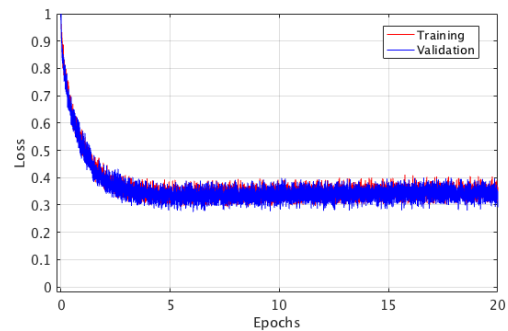
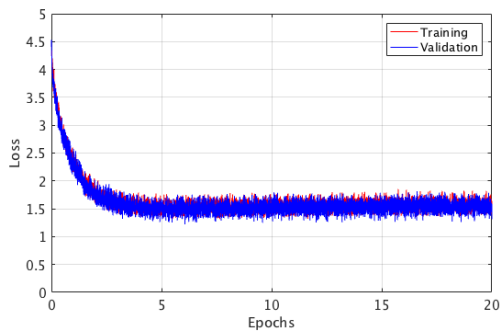
# B

## Two hyperparameter trials

The two following trials showcase the indications given whenever the learning rate parameter is too high or low.

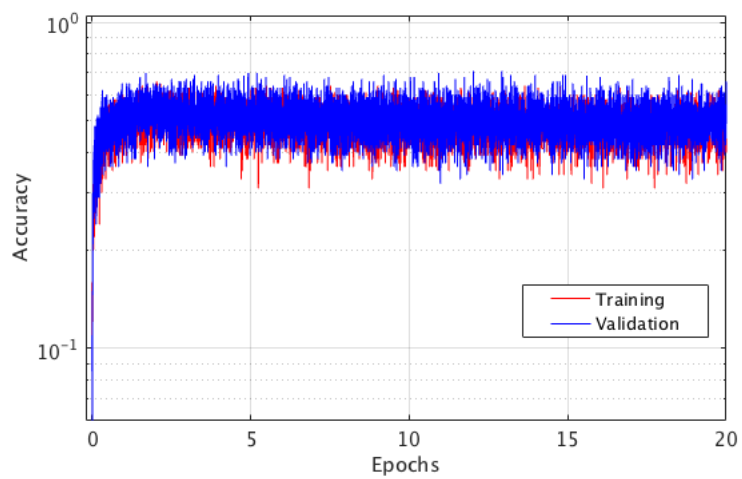
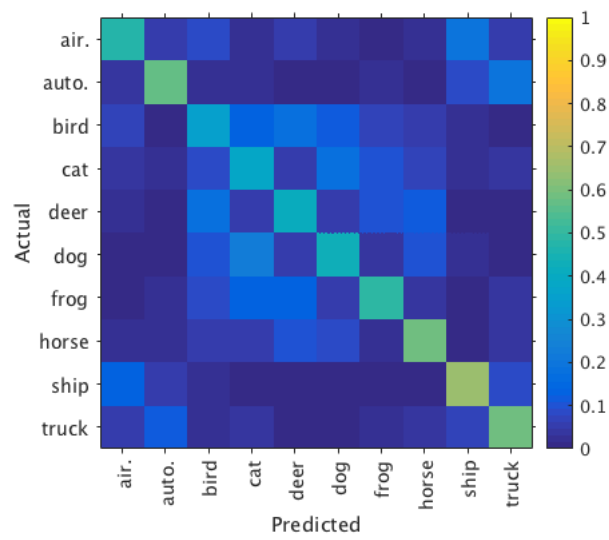
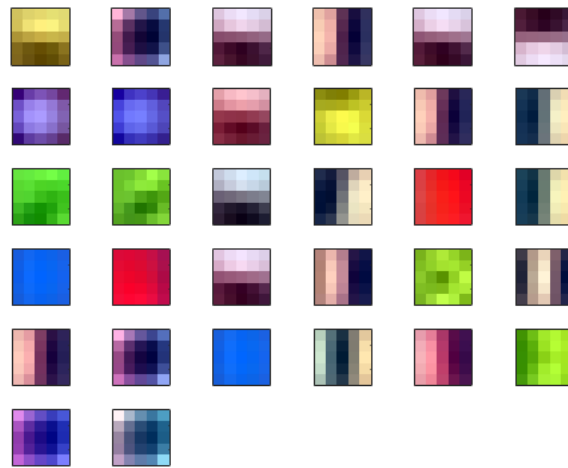
### B.0.1 Model A trial one

Hyperparameter	Value
Learning rate ( $\eta$ )	0.0001
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Mini-batch size (B)	100
Dropout rate (p)	0.5



## B. Two hyperparameter trials

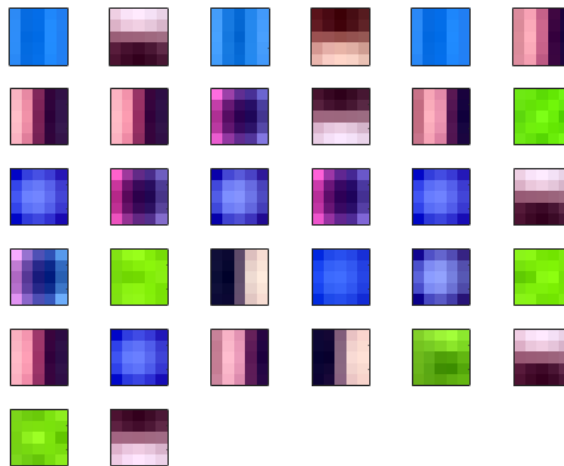
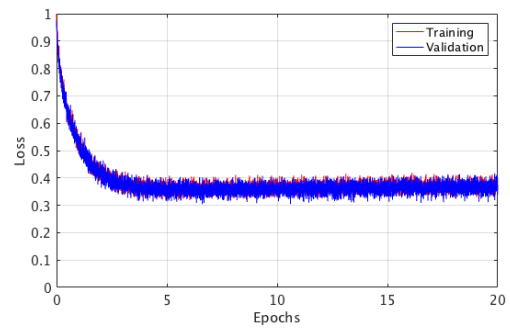
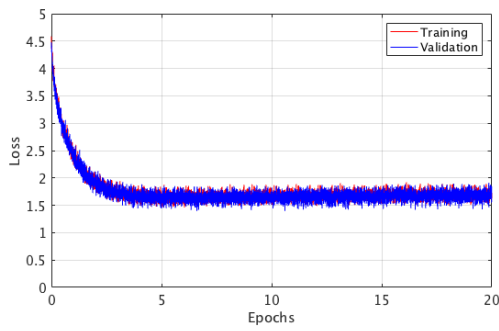
---



Test accuracy: 49.35%.

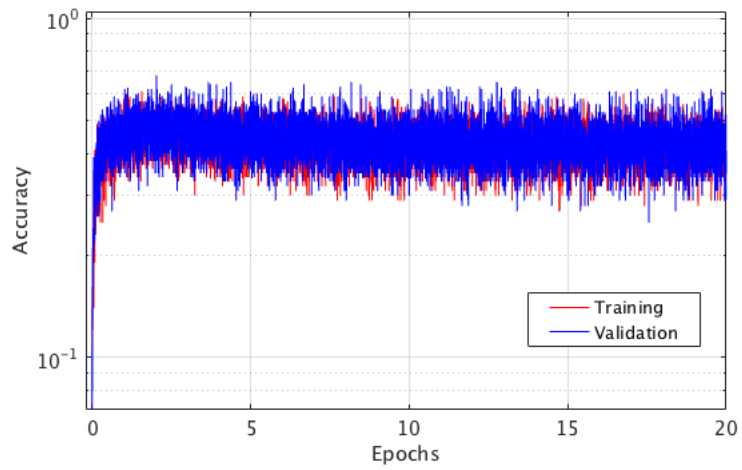
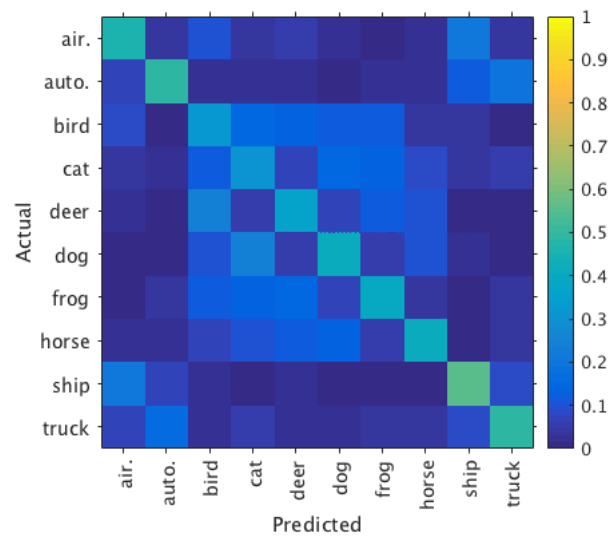
### B.0.2 Model A trial two

Hyperparameter	Value
Learning rate ( $\eta$ )	0.00005
Regularization ( $\lambda$ )	0.001
Decay rate 1 ( $\beta_1$ )	0.9
Decay rate 2 ( $\beta_2$ )	0.999
Mini-batch size (B)	100
Dropout rate (p)	0.5



## B. Two hyperparameter trials

---



Test accuracy: 43.10%.