universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Provide an automatized training customized by adjustments for different convolutional neural networks to classify characters in a real environment based on generated data“

verfasst von / submitted by

Jonas Laux, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl. Ing.)

Wien, 2018/ Vienna 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas
Mitbetreut von / Co-Supervisor:	Dipl.-Ing. Dr. Elaheh Momeni-Ortner, BSc

Erklärung zur Verfassung der Arbeit

B.Sc. Jonas Laux
Turmburggasse 1/7, 1060 Vienna

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

Wien, 26. August 2018

Jonas Laux

Danksagung

Ich bedanke mich bei meinem Betreuer Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas und meiner Zweitbetreuerin Dipl.-Ing. Dr. Elaheh Momeni-Ortner, BSc für die Unterstützung. Ich möchte auch meinen Kollegen danken, insbesondere meinem Vorgesetzten Daniel Albertini, die mich ebenfalls bei allen anfallenden Fragen unterstützt haben. Schließlich muss ich meinen Eltern und meiner Partnerin meinen tief empfundenen Dank aussprechen, dass sie mich während meiner Studienzeit und während des Prozesses der Forschung und des Schreibens dieser Arbeit unermüdlich unterstützt und kontinuierlich ermutigt haben. Diese Leistung wäre ohne sie nicht möglich gewesen.
Vielen Dank.

Acknowledgements

I would like to thank my thesis supervisor Univ.-Prof Dipl.-Ing Dr. Wolfgang Klas and my co-supervisor Dipl.-Ing. Dr.Elaheh Momeni-Ortner, BSc for all the support and guidance. I would also like to thank my co-workers, especially my work's supervisor Daniel Albertini, who also gave my great support. Finally, I must express my very profound gratitude to my parents and to my partner for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Kurzfassung

Dank modernster Technologien hat Machine Learning (ML) in den letzten Jahren die Leistungen von Computervision übertroffen. ML Prozesse erfordern ein Training. Um ein gutes Trainingsprogramm zu erstellen, das benötigt wird um ein Netzwerk funktionstüchtig zu machen, muss der für das Training verwendete Datenpool zufriedenstellend sein. Der Erwerb vorbereiteter und annotierter Daten kann jedoch kosten- und zeitintensiv sein, was durch die Generierung der Daten umgangen werden kann. Um bei solchen Aufgaben zur Vorbereitung des Trainingsaufbaus und des Trainings selbst gute Leistungen zu erbringen, bedarf es einer Kenntnis des Basiswissens. Um diese Probleme anzugehen, stelle ich ein System vor, dass alle erforderlichen Schritte in einem einfachen Ablauf vereint.

Betrachtet man das folgende Beispiel. Ein User hat einen Anwendungsfall im Bereich Optical Character Recognition (OCR) und möchte ein Modell erhalten, dass auf einer spezifischen Schriftart, welches den Kern des Datansatzes repräsentiert, trainiert ist. Durch die Modifikation dieses Kerns, der aus den extrahierten Zeichen der bereitgestellten Font-Datei besteht, wird ein Datenset erstellt, dass bei einem Training des neuronalen Netzwerks verwendet wird.

Die Parameter für den optimalen Ablauf des Trainings, sowie die Parameter der Generierung der Trainings-Datensätze werden in dieser Arbeit untersucht. Darüber hinaus wird die Architektur, Struktur und Leistung des Netzwerks angepasst, so dass es in der Industrie verwendet werden kann - zum Beispiel in Mobiltelefonen. Um einen Vergleich der verschiedenen Ansätze zu gewährlisten, evaluiere ich meine Methoden, indem ich ein

Validierungs-Datenset erstelle und jedes erstellte Modell mit diesem Set teste. Die daraus resultierende Genauigkeit wird zur Bewertung verwendet.

Abstract

Thanks to state-of-the-art implementations ML has out-performed computer vision tasks in recent years. ML processes require training. To create a good training program to enable a network to perform, the pool of data used for the training must be satisfying. However, the acquisition of prepared and annotated data can be a costly and time-consuming task. This can be circumvented by generating the data. To perform well in such tasks for the preparation of the training setup and the training itself, one requires a knowledge of the underlying processes.

To address these issues, we present a system which combines all the required steps in one simple workflow. Consider the following example. A user has a use case in the OCR scope and wants to receive a model trained on a specific font, which s/he uploads to present the ground truth. By modifying the ground truth of the extracted characters from the font file provided, a data set is created which is used in the training of the neural network model.

The training's hyperparameters and the parameters of the augmentation are studied so that the training performs optimally. In addition, the architecture, structure, and performance of the network is adjusted so it can be used in production in the industry—for example, by mobile phones.

We evaluate our methods by creating a validation set and testing every model against this to determine the accuracy the model is scoring on the set.

Keywords. Convolutional Neural Networks, Deep Learning, Data Augmentation, Optical Character Recognition, Tensorflow

Contents

CHAPTER 1

Introduction

1.1 Motivation

The need of the industrial market to automate various working steps is growing rapidly. One large focus is to increase the performance of a company's environment by outsourcing specific tasks to machines. Modern machine learning is implemented in our daily lives in applications which include face detection [GD04], IoT [GBMP13], traffic prediction [LDK+15] and data mining [WFHP16]. Even very complex tasks are now using machine learning, which can outdo humans. Consider autonomous driving, for example [FCNL12]. Machine learning is evolving quickly, and the state-of-the-art is changing year by year. Deep learning [LBH15] evolves machines to generate better results in a short time. However, for a complex problem to generate an accurate model, which can deliver good results for its use case, deep learning requires valid training. There are two different kinds of training: supervised and unsupervised. Unsupervised training is machine learning without predefined classes or goals for the model. It attempts to detect patterns of the input and adjust its learnable parameters accordingly. This approach does not need labelled data. Supervised training [KZP07] is exactly the opposite: It provides a certain type of annotated or labelled input and classifies it to a predicted output. During the training, the model or network sees only a limited number of classes, which it can classify.

This approach is tailored for a specific use case. Therefore, the problem is to get the labelled input to provide such a training. Depending on the complexity of the application, thousands or even hundreds of thousands of labelled classes are needed [GBC16].

One solution to this problem is to take one of the many free databases [Dee06] of annotated data and integrate them into the training. Several such databases are provided for different use cases. This approach can easily provide enough training material to start the training for general models. The problem with data from databases, however, is that they limit the variety of classes by the provided datasets and the training will not work if the initial problem differs from the annotated data provided.

Another strategy is to create and annotate a small dataset manually. The large downside of this approach is in the amount of data that is produced. This can lead to an unsatisfactory training, as the model may not encounter enough variance to classify reliably. In addition, manual annotating can be very complex. For example, in general object recognition, it is relatively easy for a human to annotate data. But if the use case is more complex and includes the know-how of experts in the medical domain, for example, this task can be too time-consuming and therefore too expensive.

Another approach to gaining a satisfying amount of data for deep learning training is data augmentation[All01]. The idea is to generate enough data from a small amount of data by including enough variance to satisfy the problem. Variance generation is achieved by applying various manipulations to the available data. One advantage of this approach is the size of the dataset, which can grow infinitely. In addition, the implementation of different manipulations can cover a wide range of possible deviations from the ground truth. Otherwise, a disfigured generated dataset can overfit the model so that it cannot classify the original use case anymore. The adjustments of the hyperparameters to the data augmentation are critical for the training pipeline. Nevertheless, random initializing of the hyperparameters and manipulations or even the ground truth is commonly used in data augmentation. But more sophisticated attempts which use predefined parameters for synthetic (generated) data have also performed well in trainings. As an example, a hyper-realistic image manipulated with a defined set of mutations can satisfy training for

an object detector with good results [RLF15].
A crucial part of the implementation of this system is the architecture of the network. The composition of the network should deliver satisfying results in an acceptable amount of time. Yann LeCun created the LeNet-5 [LBBH98] in 1998, which is a network that performs well with handwritten characters. The minimal structure of the LeNet-5—which consists of two convolutional layers (Section 2.2.1), two pooling layers (Section 2.2.2), and two fully-connected layers (Section 2.1.2) is a good choice performance wise, and it scores acceptable results (>90%). Next, various network structures were created to meet a similar challenge with a decreased error rate and greater accuracy with respect to a larger pool of data sets. AlexNet [KSH12], ResNet [HZRS16], and GoogleNet [SLJ+15] are some examples of deep neural-network architectures that have achieved good results. The problem is that these architectures are too deep and too costly with respect to performance for an application which is to run on mobile phones.

1.2 Problem Statement

Currently, to our knowledge, there is no automated script or development which can automatically provide training for a Convolutional Neural Network (CNN) in the OCR domain and which can afterwards classify out of a limited specific pool of characters in a real environment based only on generated data.

1.3 Goal of the thesis

This thesis addresses the problem described above. The first goal is to provide a solid training for the classifier mentioned in Section 1.2. In addition, the amount of data generated will be studied with respect to the end result and to determine whether a higher amount of data influences accuracy. The latter is to be measured by one validation set taken with examples from a real environment.
The second part of this report offers an evaluation of a solid architecture for the CNN.

The augmentation should also fit different architectures so it can be adopted by other use cases.

1.3.1 Scope

The scope of this work is the research of a CNN architecture and training methods. Furthermore, the implementation of a pipeline for the training for the network which is specified by the input of a user. This network shall classify characters (detection is not part of this work). The input is a font file (*.ttf*), to which the network are to be particularized. With the given ground truth, a satisfying amount of data will be created. To create a good user experience, a frontend is implemented to upload the user's font-file. A Python web-Server is used to delegate everything.

1.4 Overview of the thesis

Chapter 2, provides an overview of the basics of ML that are used in this thesis, which include deep learning, Neural Network (NN) and data-augmentation algorithms. Chapter 4 discusses the methodology, which includes the design of the CNN and the manipulations that are used for the augmentation. Chapter 5 describes the exact implementation of the design and training of the CNN described in Chapter 4. The mutation implementation of the data augmentation is also covered here. In addition, the Python server, which runs all the tasks, is explained. Finally, Chapter 7 collects, sums, and evaluates the obtained results and Chapter 7 provides a summary of the thesis and offers some ideas for future work.

CHAPTER 2

Background

2.1 Artificial Neural Network

2.1.1 Perceptron

Created by Frank Rosenblatt in 1958, Perceptron is the first-ever artificial neural networkArtificial Neural Network (ANN) [Ros58]. The idea was to mimic human learning artificially. A neuron's dendrites are simulated by weights and multiplied by input values. The multiplied values and the bias, which are added to model a neuron's required activation potential, are summed up in the cell body. The sums are passed through the activation functions to produce an output. This architecture is depicted graphically in Figure 2.1.

The Perceptron approach is a linear function, which means that it cannot represent more complex functions such as an *XOR* function.

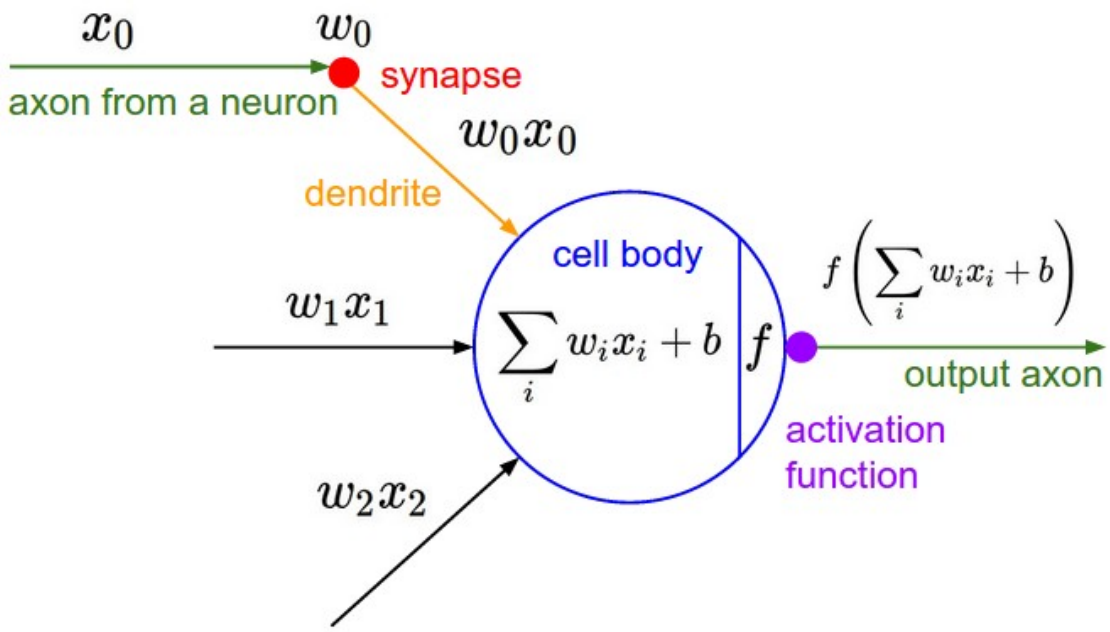


Figure 2.1: Perceptron architecture with weights w_i, inputs x_i and bias b, from [And16a]

2.1.2 Neural Network

A single perceptron has limited capabilities. To extend this, NNs or ANNs were developed. These connect multiple perceptrons, also called neurons or units, to acyclic graphs. This means that some neurons are connected to the previous or next layer but never in a cycle, which would lead to an infinite loop. The most common structure uses multi-layer perceptrons to organize the neurons into layers. Different types—which include inputs, outputs or hidden layers—are explained below. These layers increase the complexity and size of a network architecture. An example of a multi-layer perceptron is depicted in Figure 2.2. Normally, the layers are fully connected. This means that all the neurons of one layer are connected to every neuron of the previous or next layer. Neurons in the same layer are not connected at all, as can be seen in Figure 2.2.

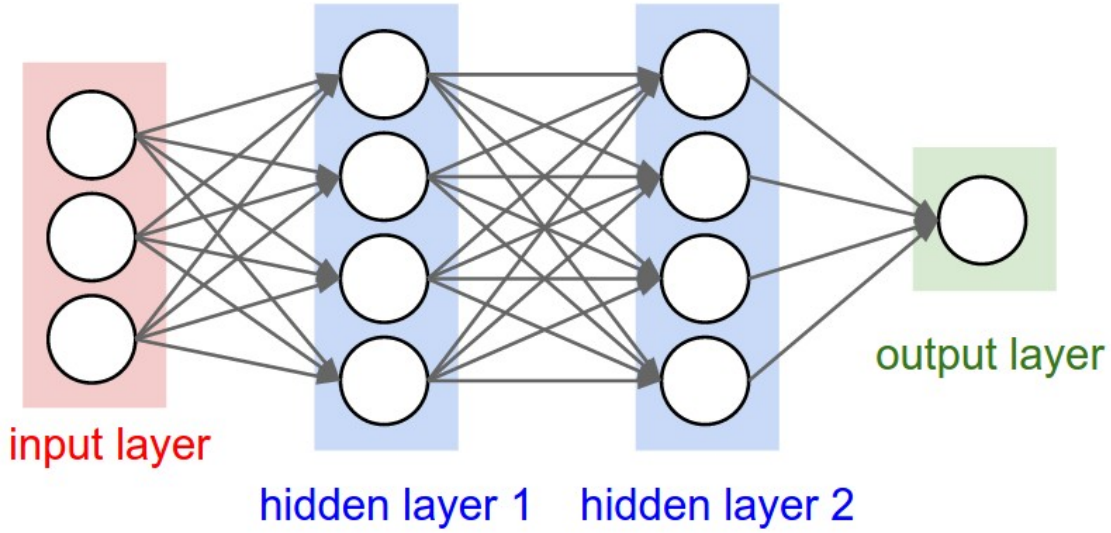


Figure 2.2: 3-layer (the input layer does not count) architecture of a NN with two hidden layers containing four neurons each, input and output layer, from [And16b]

In Figure 2.2, we see the architecture of an NN. Every NN has learnable parameters which are adjusted during training sessions. The parameters contain biases and weights. For example, there are 41 learnable parameters of the network depicted in Figure 2.2. Three layers are shown in the network of Figure 2.2 (without the input layer). These three layers contain nine neurons:

$4 + 4 + 1 = 9$ neurons

Dot product of all layers (input layer not counting) to get the weights:

To get the weights, the dot product of each layer (input layer not counting) with the input and the designated output is calculated and summed up:

$3 \cdot 4 + 4 \cdot 4 + 4 \cdot 1 = 32$ weights

Each neuron has a bias, which is also added: $4 + 4 + 1 = 9$ bias

This will result in 41 learnable parameters.

Every network has many sensitive parameters in his architecture, which must be chosen for the best fit of the use case. The number of neurons per layer and the number of layers in a network are sensitive parameters. Both can increase or decrease accuracy and efficiency. For example, too many neurons or layers can result in a loss function which is

too complex and which will overfit (Section 2.1.4) the network to the training set. On the other hand, too few neurons or layers can produce a simple function which will not classify an application with complex data.

2.1.3 Activation functions

Activation functions are used to perform mathematical operations (depending on the activation function) on the output and then decide if the neuron is "activated" (sending its value to the next layer). As an example, all layers in a network should receive only values >= 5. A simple neuron in a layer has a value which is calculated with $Y = \sum(weights + input) + bias$.
The value of the function Y could be anything from $(-\infty, \infty)$. In this case, the activation function sends (activate) values to the next layer only if they are between $(5, \infty)$. Rectified Linear Unit (ReLu) is typically used in CNNs [KSH12]. Other common activation functions include Tanh, Sigmoid, Leaky ReLu and Maxout.

Rectified Linear Unit

A similar approach is the ReLu activation function, which thresholds the output at zero, as seen in Figure 2.3 and described in Equation (2.1).

$$\sigma = f(x) = max(0, x) \tag{2.1}$$

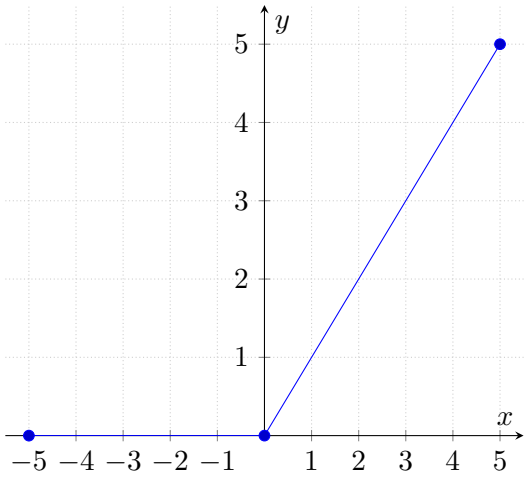


Figure 2.3: Example of a Rectified Linear Unit activation function

Softmax

The Softmax function is a logisitic function, which is simply a multidimensional generalization of the sigmoid function. It takes the output of the network, which creates a prediction of every class it is to classify. To become actual probabilities, the result has to sum up to one [Nie18].

$$\begin{bmatrix} 0.43 \\ 0.28 \\ 0.19 \\ 0.88 \end{bmatrix} \qquad \begin{bmatrix} 0.24 \\ 0.20 \\ 0.19 \\ 0.37 \end{bmatrix}$$

(a) Input of the Softmax function. (b) Output of the Softmax function.

Figure 2.4: Example of a classification of 4 classes before and after the Softmax.

2.1.4 Optimizing/Training

The goal of the training is to minimize the error of the network. To reach this goal, the hyperparameters, which consist of weights and biases, are adjusted to score the predicted output y given the input x.

Loss Function

To measure how well a network is classifying the input, a loss function J (often also referred as cost or error function) is defined [GBC16]:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(x_i, y_i, \theta) \tag{2.2}$$

Given a set m of training examples $x_i = [x_1, x_2, ..., x_n]$ and their predicted output $y_i = [y_1, y_2, ..., y_n]$, $J(\theta)$ is calculated from the per-example loss function $L(x_i, y_i, \theta)$, where θ is defined as combination of network parameters which consists from the collection of all weights (w) and biases (b).

For example, the per-example loss function L may be defined as follows:

$$L(x_i, y_i, \theta) = ||a(x_i, \theta) - y_i||^2 \tag{2.3}$$

Where $a(x_i, \theta)$ is the output of the network. The example above is the Mean Square Error (MSE) loss function.
Smaller values of loss function typically lead to a better prediction of the network. This is why training is often defined as optimizing the loss function $J(\theta)$. To decrease the value, the gradient descent method (Section 2.1.4) is a common way to find a minimum of the loss function.

Gradient Descent

The training aims to find a local minimum of the loss function, as described in Section 2.1.4. If we have a simple loss function, as shown in Figure 2.5, calculating the local or global minimum seems affordable [Sch15].

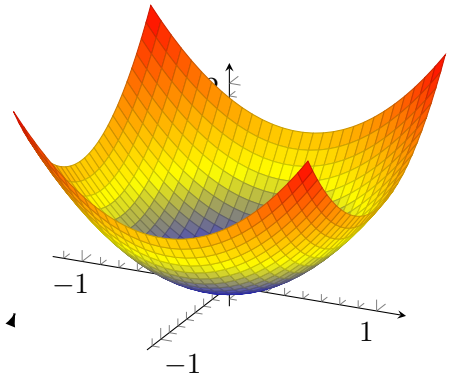


Figure 2.5: 3D Error function showing the global minimum of a convex loss function.

If the loss function $J(\theta)$ is more complex—for example, as in a CNN—millions of parameters can be adjusted. Such adjustment makes finding a minimum of the loss function much harder [LBH15]. One way to approach a minimum is via gradient descent. Here, randomly starting parameters are initialized and repeatedly going to the steepest descent, as shown in Figure 2.6.

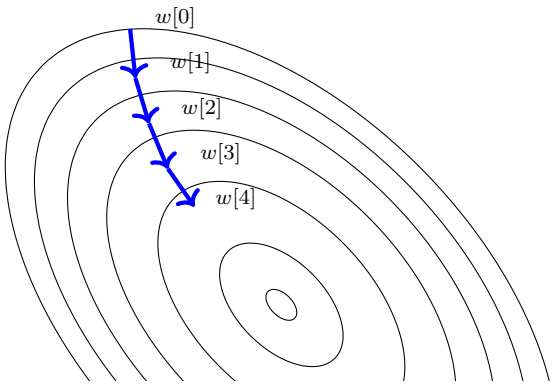


Figure 2.6: Gradient descent through multiple iterations finding a minimum.

Computing the direction of the steepest decent at an iteration of the training x_i with the output y_i, and respecting different parameters θ, is achieved by calculating the negative gradient [GBC16]:

$$- g_i = \nabla_\theta L(x_i, y_i, \theta)) \tag{2.4}$$

To identify the final gradient for the training, an average of all gradients [GBC16] can be determined:

$$- g = \frac{1}{m} \sum_{i=1}^{m} -g_i \tag{2.5}$$

The algorithm depends on the start values which are taken to find a global minimum. This is possible, but it sinks with the increase of complexity. Thus, finding the global minimum is not guaranteed if the loss function is not convex, as in Figure 2.5. One critical parameter for applying the gradient descent to a loss function is the size of the period calculating it for different parameters of θ. This parameter is called the learning rate ϵ.. This leads to a gradient-descent update rule [GBC16]:

$$\theta = \theta + \epsilon \cdot -g \tag{2.6}$$

The approach to finding a minimum for a loss function $J(\theta)$ with respect for its parameters θ is straightforward. The approach to finding loss function for a CNN, including all the weights and biases is not straightforward, as it is necessary to determine the influence of those changes. Therefore, backpropagation (Section 2.1.4) can be used.

Initialization of Weights and Biases

As described in Section 2.1.4, all parameters are initialized before the training is stepping in every iteration to the steepest descent, where one step has the length of the learning rate ϵ. The initialization can be a crucial part of the training. For example, if neurons in the same hidden layer have the same initialized weights, they always receive the same gradient; thus, there is no source of asymmetry between neurons [And16c].
So that they do now exhibit a large variance between the weights, they should be initialized near to zero but not exactly zero. In addition, they should be unique so as to

compute distinct updates and integrate themselves as diverse parts of the full network [And16c].
Biases are commonly initialized at zero [And16c].

Backpropagation

After the learnable parameters of a network have been initialized, the classification is no better than a random guess, as the weights and biases are not yet trained to anything. To increase the accuracy for deep networks, backpropagation can be applied iteratively. Backpropagation was first introduced in the 1970s [Sch15]. Combining this idea with the learning of NNs was popularized in 1986 [RHW86] and later in 1992 [HN92]. In 1989, backpropagation was applied to weight-sharing convolutional neural layers with adaptive connections [Sch15]. It has become the most essential ingredient of modern deep learning approaches. To break down an algorithm, the first step is to calculate the *forward propagation* which involves calculating the network's output iteratively from the input layer to the output layer. Thus, the output from each layer is computed with activation functions a and with each output without activation functions z and the result of the final layer D as a^D. It is important to differentiate the output before and after the activation functions, as the output of the network depends directly on the weights of the input (except activation functions) (Section 2.1.3). Next, the loss function $J(a^D, y_i)$ can be calculated, respecting the expected given truth of the output y_i.
To begin the explanation of the backpropagation algorithm, note that the matrix notation is elucidated first of all. Equation (2.7) depicts a weight matrix of one layer shown. The w^l_{jk} describes the weights which connect the neurons j^{th} from the previous layer $(l-1)$ to the neurons k^{th} in the current layer l. With these weights w, we can create the weight matrix W^l where the value in row j^{th} and column k^{th} is w^l_{jk} and the number of M neurons from $(l-1)$ and N neurons from l:

$$W^l = \begin{bmatrix} w^l_{11} & \dots & w^l_{1N} \\ \vdots & \ddots & \vdots \\ w^l_{M1} & \dots & w^l_{MN} \end{bmatrix} \tag{2.7}$$

Every other relevant element—such as biases b^l_j and the output of a layer before z^l_j and after a^l_j the activation function of neuron j—can be represented as vectors b^l, z^l and a^l:

$$b^l = \begin{bmatrix} b^l_1 \\ \vdots \\ b^l_M \end{bmatrix} \qquad z^l = \begin{bmatrix} z^l_1 = \sum_k w^l_{jk} a^{(l-1)}_k + b^l_j \\ \vdots \\ z^l_M = \sum_k w^l_{jk} a^{(l-1)}_k + b^l_j \end{bmatrix} \qquad a^l = \begin{bmatrix} a^l_1 = \sigma(z^l_1) \\ \vdots \\ a^l_M = \sigma(z^l_J) \end{bmatrix} \tag{2.8}$$

Contrary to *Forward propagation*, *Backward propagation* starts with the calculated gradient g^D_a of the output from the final layer D [Nie18]:

$$g^D_a = \nabla_{a^D} J(a^D, y_i) \tag{2.9}$$

To obtain the gradient of the output without the activation functions, we calculate the entry-wise product of the gradient of final output g^D_a and the derivative of the activation function σ' from the output layer D [Nie18]:

$$g^D_z = g^D_a \odot \sigma'(z^D) \tag{2.10}$$

To calculate the gradient of the output of the previous layer after the activation functions, the transposed weight matrix $(W^D)^T$ is multiplied with the gradient g^D_z computed above [Nie18]:

$$g^{D-1} = (W^D)^T g^D_z \tag{2.11}$$

To receive the final gradient for this one single example, the algorithm above is calculated iteratively through the network's architecture [Nie18]:

$$g\theta = \nabla_\theta J(\theta) \tag{2.12}$$

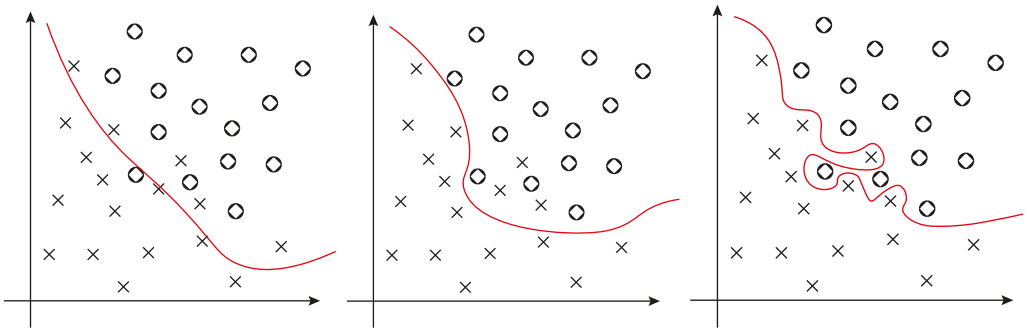
Thus, the gradient descent updates the parameters θ of the objective $J(\theta)$ over the complete dataset [Nie18]:

$$\theta = \theta - a\nabla_\theta J(\theta)) \tag{2.13}$$

With this algorithm, it is possible to generate a gradient for a single training example, which then can be used to justify the learnable parameters of the network [Nie18][LBH15].

Regularization

As is explained in this section, ANNs are trained with gradient descent and backpropagation to make better predictions. The training uses examples from a data set. The problem is that this method does not guarantee that the network can classify with equal accuracy examples that are not from the data set. Therefore, a common approach is to split the data set into validation, test and training sets. These are disjoined from each other such that every example is unique to the whole data set. The test set is usually used to evaluate the model with respect to unseen examples during the training. If the network performs worse on the test set than on the training set, this is called *overfitting*. This means that the loss of the training set is much less than the loss of the test set. The opposite is called *underfitting*, which occurs when the test loss is smaller than the training loss [LBH15].



(a) Underfitting (b) Appropriate fitting (c) Overfitting

Figure 2.7: Abstract classification example of underfitting, appropriate fitting and overfitting.

In Figure 2.7, are all three stages. Underfitting is normally not a problem, as the architecture of the NN is commonly not large deep enough or the hyperparameters of the training are not well chosen with respect to, for example, the length of the training (iteration count) or the amount of training data.
Overfitting is loosely said, that the network is trained too intense on the training data, so it can only classify those. This can be prevented, for example, by *dropout* [SHK+14] or by briefer training.

2.1.5 Stochastic gradient descent

Another method used to automatically train networks is called Stochastic Gradient Descent (SGD). As described in Equation (2.13), the standard gradient-descent algorithm updates the parameter θ in $J(\theta)$ over the whole training set. The SGD simply calculates the gradient of the parameters using a limited amount of training data called a batch. This is also helpful if the data set is too large for the hardware to handle when the training is running.
In every iteration of the training, instead of taking the whole training set, a sample is chosen: $B = [x_1, x_2, ..., x_{m'}]$ where m' is the size of the batch B. The loss function then uses the batch instead of the whole training set [GBC16]:

$$J(\theta) = \frac{1}{m`} \sum_{i=1}^{m`} L(x_i, y_i, \theta) \tag{2.14}$$

Batch normalization

To guarantee that the input of a layer is not randomly scattered or sized, which would affect the training, the input is normalized [IS15]. The problem is that if the complete input is normalized for every layer in every iteration, the calculation of the gradient descent (Section 2.1.4) cannot be done practically, as it would be too inefficient and costly. In contrast to standard gradient descent, the SGD calculates only the gradient of the loss function $\nabla_\theta J(\theta)$ for a small number m of randomly chosen examples, called a batch [IS15]. The batch is taken out of the test set and is chosen only one time. After the calculation of the loss function, the next randomly chosen batch will contain completely different examples. This allows for training with higher learning rates and fewer iterations [IS15].

Adam Optimizer

Adaptive Moment Estimation (Adam) is a SGD used for computing the learning rate which is adaptive for each parameter [KB14]. Adam stores the decaying average of past squared gradients v_t and the exponentially decaying average of past gradients m_t [Rud16]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)(\frac{1}{m} \sum_{i=1}^{m} g\theta_i) \tag{2.15}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\frac{1}{m} \sum_{i=1}^{m} g\theta_i)^2 \tag{2.16}$$

Here, β represents the momentum term, which is another hyperparameter that can take a value from zero to one. t indicates the current training iteration. m_t and v_t represent the first and second momentums of the gradient, which are initialized as vectors of zeros, [Rud16], the author of Adam [KB14], observes that those momentums are biased towards

zero—especially when they are initialized and when the decay rates are small (i.e. β_1 and β_2 are close to one). Next, these first and second momentums are bias-corrected:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.17}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.18}$$

These momentums are then used to update the parameters:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t \tag{2.19}$$

where η is a term to avoid division by zero.

2.1.6 Transfer Learning

A commonly used approach is not to train a CNN from scratch but to use a well-trained network and retrain it on the own purpose [PY10] [YCBL14]. The base features and patterns of the classified objects are mostly equal to the specific use case. For example, the result of a classifier, that must classify characters for mostly every font equal in the basically shape. The variation of the basically shape of the letter "A" will normally not be very high in different fonts. There are two different ways to use transfer learning.

1. **Using the pre-trained CNN's weights without adaption.**. Every weight of the pre-trained model is implemented in the new CNN. Only the fully-connected layers are replaced and optimized to the specific use case.

2. **Using the pre-trained CNN's weights with small adaptions.**. The weights are also implemented but are not ignored in the training. They are fine-tuned by continuing the backpropagation. This approach can also be adapted to the specific use case. Either every layer can be fine-tuned or just the higher layers with the lower layers fixed. For example, if the pre-trained model can classify animals but

the use case involves classifying a specific horse breed, then the layers with the higher features can also be retrained but the layers which classify the base features of a horse can be used as they are.

2.2 Convolutional Neural Network

One downside of a regular NN is that every layer is fully connected. In the example above, the number of learnable parameters is manageable. However, if the size of the input increases, the number increases quickly. For example, if we have an input image of 200x200x3 (200x200 pixel resolution and three colour channels), we end up with 120,000 weights for only one layer. This fully-connected approach leads to a very large number of wasteful parameters, and the risk of overfitting increases dramatically.

A CNN is a specific type of a ANN [And16c]. The idea is to add a dimension to every layer. Instead of having fully-connected layers, which consist of width and height (as shown in Figure 2.8 as an input with 32x32 pixel resolution), a convolutional layer contains also depth.

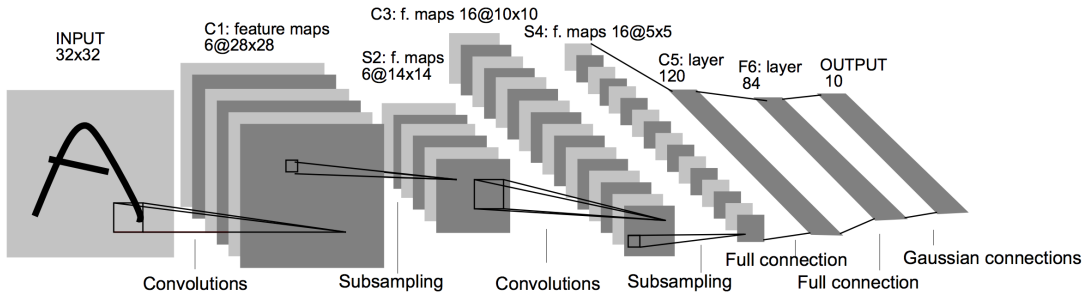


Figure 2.8: Architecture of a CNN called LeNet-5 including convolutional and pooling layers, from [LBBH98].

2.2.1 Convolutional layer

Usually, the input of a CNN is a two- or three-dimensional matrix. These can be images (2D for grayscale, 3D for RGB images) and may represent the pixels. Multiple filters are created. These filters (also called kernels) are also matrices. These must have the

same depth as the input matrix. The height and the width can differ with respect to the input. The size of the filter depends on the use case. If complex objects are classified, the input image will have a higher resolution, which decreases the chance of losing features or patterns. If the structure of the objects is more basic, smaller filters can be chosen. For an RGB image with a resolution of 50x50x3 (50px wide, 50px high, 3 colour channels), a filter could have the structure of 3x3x3. This filter contains the weights (the learnable parameters: in this example, 27. These weights can be initialized randomly, but there are also approaches to doing the initialization (Section 2.1.4) in a structured fashion. The different channels of the filter can contain different weights. For example, a simple edge detection for every channel can look like this:

$$\begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

(a) Red channel filter (b) Green channel filter (c) Blue channel filter

Figure 2.9: Detection filter of vertical left-side edges for every channel of an RGB image.

If the use case specifies to detect only blue edges, the filters of the other channel would look different.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} -1 & 1 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

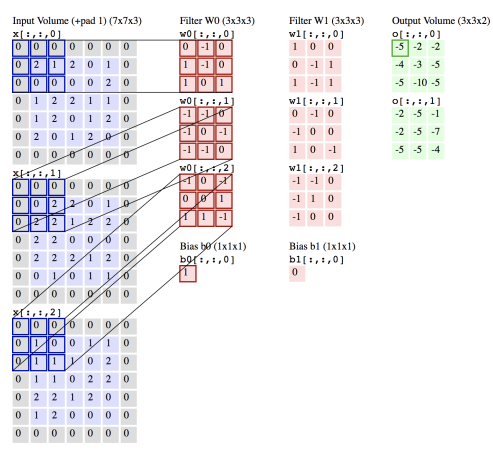
(a) Red channel filter (b) Green channel filter (c) Blue channel filter

Figure 2.10: Detection filter of vertical left-side edges for the blue channel of an RGB image.

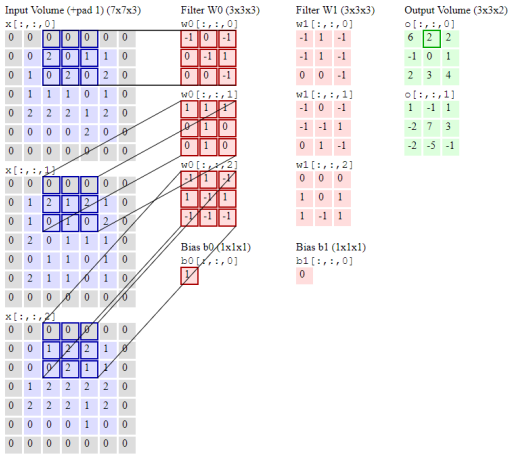
This example of an edge-detection filter is usually learned by the network itself and is not set previously. The number of filters per layer also depends on the complexity of the input and can differ heavily between different use cases. As a rule of thumb, a good way to go is with the best result the network can score with the fewest parameters. Gaining complexity of the network by increasing the layers, filters or weights per filter

can improve the result, but it can also decrease the performance and increase the risk of overfitting.

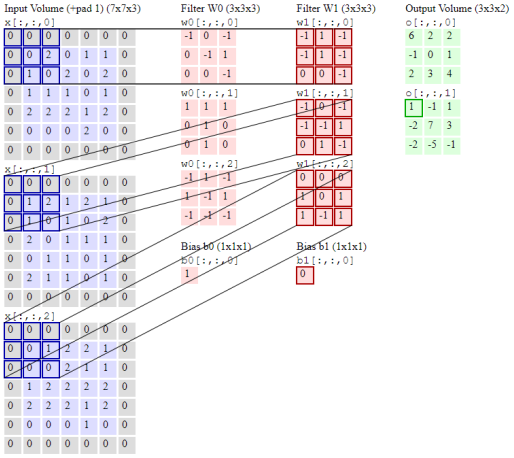
The workflow of a convolutional layer is to shift the filter over the input value by value. The amount of values the filter is shifting between each step is called the stride, and it can be customized. If the input and the filter are multi-dimensional, every channel of the filter is used in parallel.



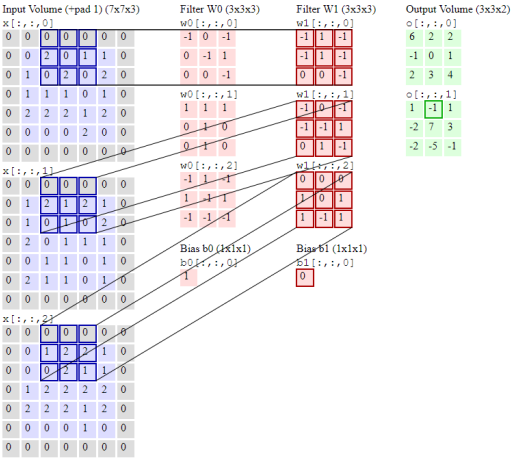
(a) The first step in a convolutional process for the first filter $W0$ with three dimensions.



(b) The second step in a convolutional process for the first filter $W0$ with three dimensions.



(c) The first step in a convolutional process for the first filter $W0$ with three dimensions.



(d) The second step in a convolutional process for the first filter $W0$ with three dimensions.

Figure 2.11: Workflow of a convolutional layer, from [And16c].

In Figure 2.11, the first steps of the convolutional process is shown (in this example, a padding is chosen). For every channel in x, the first filter $w0$ obtains the local area of the first value of the input x, including the padding. The dot product of the two matrices x and $w0$, from the matching channels, is calculated. The dot product of every channel are summed up, which will then be the first value of the first channel of the output volume o. For the second value, which is calculated in Figure 2.11b of the first channel from the output o, the local area of the input volume shifts over two values (size of the predefined stride) and calculates it. This is done by every filter that is included in the layer (second filter workflow in Figure 2.11c and Figure 2.11d), and it generates a new channel of the output o.

Padding

One problem that occurs with shifting by the amount of the stride is that the spatial dimension decreases with every convolutional layer.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{3x3 filter}} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.12: Convolutional operation with no padding, 5x5 input, 3x3 filter, a stride of 1x1 and a 3x3 output

As seen in Figure 2.11, the filter will start at the top left corner and shift over to the right with a stride of one. This produces three shifts, until the filter has to start again one row below. If we consider a simple situation as shown in Figure 2.12, the convolutional workflow will decrease the size of the input volume. The constant decrease of the volume leads to a loss of information by adding new convolutional layer. Especially in the first layer, this information is needed to extract low-level features. To add more convolutional layers and still preserve the information, a zero padding is added to the input.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{3x3 filter}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 2.13: Convolutional operation with a zero padding of one, 5x5 input, 3x3 filter, a stride of 1x1 and a 3x3 output

In Figure 2.13, one line of zero padding is added. With this, the convolutional operation can be performed without spatial size decrease and information loss.

2.2.2 Pooling layer

A pooling layer reduces the spatial size of an input to decrease the computation and the risk of overfitting. It is common to put these between successive convolutional layers periodically. The size of those layers is rather small, to reduce the risk of losing patterns. A common size is 2x2. The pooling algorithm takes the size of the first input region and uses a pooling function: for example, the MAX function. Other functions such as average pooling or L2-norm pooling can also be performed, but these do not show better results than the MAX function in most cases.

$$\begin{bmatrix} 4 & 2 & 3 & 1 \\ 1 & 5 & 2 & 0 \\ 4 & 1 & 2 & 1 \\ 0 & 2 & 6 & 7 \end{bmatrix} \xrightarrow{\text{Max pooling with 2x2}} \begin{bmatrix} 5 & 3 \\ 4 & 7 \end{bmatrix}$$

Figure 2.14: Using a max pooling function on a 4x4 matrix.

These pooling layers must not be used. There are ways to use larger strides on convolutional layers to decrease spatial size instead of pooling [SDBR14].

2.2.3 Normalization layer

Every layer's output affects the next layer. This effect brings the problem of internal covariate shift. If layers closer to the output of a deep network are adapted by the input of lower layers, those lower layers are also adapted, which leads to the parameters of the higher layers becoming worse. One solution to this problem is to use batch normalization [IS15]. For example, if the layer to normalize has d dimensions $x = (x_1, ...x_d)$, the kth dimension normalization would look like this:

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}} \tag{2.20}$$

The normalization must be scaled and shifted, because it would otherwise limit the layer's representation. To prevent this, the normalization input $\hat{x}$ is adapted as follows:

$$\hat{y}^k = \gamma^k \hat{x} + \beta^k \tag{2.21}$$

where γ and β are learnable parameters.

CHAPTER 3

Related Work

In 1989, Yann LeCun was working on creating a network to recognize hand-written digits and created the idea of convolutional neural networks [LCJB+89]. As a pioneer work, he designed the LeNet-5, which he designed on a dataset on hand-written zip codes from a U.S. post database. The LeNet-5 was capable of sorting mail by zip codes with an error rate of only 5% [LBD+89] and so became one of the first models to replace human interactions with ML. This shows that CNNs are a good fit for OCR [LBBH98]. In 2012, Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever won the ImageNet Large Scale Visual Recognition Challenge (LSVRC) with their network architecture called AlexNet. They trained it with the 1.2 million high-resolution images from the ImageNet LSVRC in 2010 containing 1000 classes. It's error was only at 15.3%, which is 10.8% less the second-best entry [KSH12]. In 2013, Dahl et al. has improved CNNs by implementing ReLu (Figure 2.3) and Dropout [DSH13]. As in 2015 the idea of batch normalization 2.1.5 was introduced. Batch normalization is stabilizing CNNs more than ReLu or Dropout [IS15].

One interesting way to gain performance by implementing CNNs in mobile phones is to create mobile nets. Mobile net is a term for efficient models for mobile and embedded vision applications [HZC+17].

The importance of data augmentation (Section 4.1), which is the main topic of this thesis, was discussed in 2014 [PRH+14]. Multiple transformations and mutations are explained which can increase or even decrease accuracy.

As a mirror of the pre-work from [LCJB+89], some systems with a highly flexible learning schema attempt to learn a lot from labelled data with a minimum of knowledge. Multi-layered neural networks have been applied in the character scope and are competitive with other leading methods [SG07]. One of the most successful methods is Tesseract, which is an open-source OCR engine developed by HP. It uses adaptive thresholding and connected component analysis to recognize characters [Smi07]. As shown in [KSK+16], CNNs outperform Tesseract in accuracy and performance.

CHAPTER 4

Methodology

4.1 Data Augmentation

One of the most important prerequisites of an optimally functional network is a solid dataset for training or validation. A small dataset may not reflect all the possibilities of the use case. In addition, a network may overfit to a small amount of data, as it needs a certain number of iterations in the training. If a network sees one example too often, it adjusts its parameters to just that set of examples [All01]. Therefore, much annotated data is needed, as mentioned earlier (Section 1.1). In addition, manual annotation of the data can require advanced knowledge—as in a medical domain, which also results in a much greater workload. With data augmentation generating different mutated data out of existing data, the downsides of a small dataset can be overcome [RDS+15]. Even with the use of a large, annotated dataset such as ImageNet [RDS+15], it has been proven that data augmentation benefits the outcome as an additional regularizer [KSH12]. In addition, situations of different disorders on the input (reflections, blurring etc. . .) are more covered by mutations and combination of mutations. This may reflect more of an impression of a real environment. For example, if the input image includes the noise of a bad camera or is cropped because of a bad detection, the network saw those edge cases and may still classify it correctly. Otherwise, the parameters of those mutations

are very sensible. Data which is too disfigured may not reflect reality, which may lead to an overfit of the network.

As a requirement for this service, the dataset is limited to only one image per class, as the dataset is generated from one provided font file which constitutes the ground truth of the augmentation. After a selection of required characters, those are extracted from the font, which results in one image per class of the shape 38x64 pixel resolution, with the white character and a black background saved in a folder named after the ASCII declaration.

4.1.1 Translations

To overcome the possibility of bad detection, the Region of Interest (RoI) is not placed in the centre of the input. These translations are along the x- and the y-axis.

(a) Original image. (b) Translated image

Figure 4.1: Translation added to the original image.

4.1.2 Rotation

The input might not be exactly straight, as it is exported from a font file. In this case, a rotation may be needed. To make sure that the character is still 100% in the image, scaling is required.

(a) Original image. (b) Rotated image

Figure 4.2: Rotation added to the original image.

4.1.3 Scaling

The RoI can appear in different sizes. The character should be classifiable with different sizes. This makes pre-processing after detection easier, as the application does not have to resize the image to one size every time.

(a) Original image. (b) Scaled image

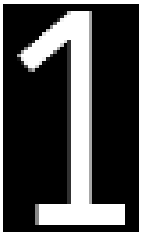
Figure 4.3: Padding added to the original image.

4.1.4 Distortions

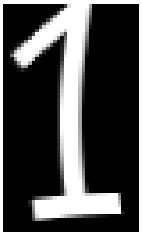
To reflect a real environment, sinus (positive and negative), barrel and pincushion distortions make sense here, as they cover the range of occurring image distortions. In addition, these are used throughout the spectrum of the image and are not always in the centre.

Sinus distortion

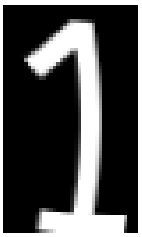
A sinus distortion covers the manipulations through a line of a character pushed to either the negative or positive x-axis.



(a) Original image.



(b) Positive sinus distorted image

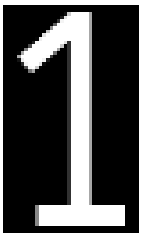


(c) Negative sinus distorted image

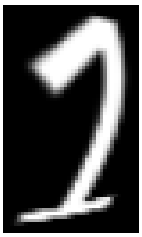
Figure 4.4: Positive and negative sinus distortion added to the original image.

Barrel and Pincushion distortion

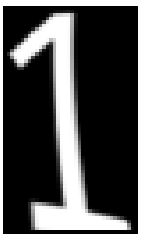
Barrel or Pincushion distortion can occur due to different lenses from the camera (e.g., fisheye) or by disorders on lenses, such as water.



(a) Original image.



(b) Barrel distorted image



(c) Pincushion distorted image

Figure 4.5: Pincushion and barrel distortion added to the original image.

4.1.5 Perspective warps

One of the most important mutations is the perspective warp. This mutation takes care that the angle the character is detected.

(a) Original image. (b) Perspective warped image

Figure 4.6: A perspective warp added to the original image to imitate a detection angle.

4.1.6 Gaussian noise

The input can always be of a lower quality than expected. To increase the probability of classifying images with random noise (e.g., added by an ISO which is too high in a dark environment), the augmented dataset must include this disorder.

(a) Original image. (b) Image with noise

Figure 4.7: Gaussian noise added to the original image.

4.1.7 Gaussian Blur

Adding blur to the range of mutations is important, as detected RoI can be taken with a camera that is not completely focused on the character.

(a) Original image. (b) Blurred image

Figure 4.8: Blurring added to the original image

4.1.8 Reflections

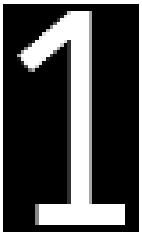
Reflections are important for imitating the real environment, as the surface the characters are on can vary. Thus, reflecting material should be considered as background. As an example, a mobile device that uses a flash for taking an image as input for the classification, where the target is printed on a reflecting material, the taken image will include reflections.

(a) Original image. (b) Image with reflection added.

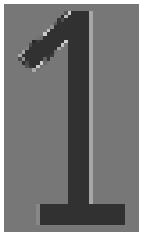
Figure 4.9: A reflection added to the original image

4.1.9 Background and foreground

To ensure that the classifier can identify different light settings and colours, the background and foreground are generated with different properties.



(a) Original image



(b) Back- and foreground transformed image

Figure 4.10: Foreground and background manipulation

4.2 Data sets

As described in Section 4.1, data is augmented to provide solid training for the network. This augmented data is used as the training and test set. Before training, the percentage of the test set is set, for example, in this approach, 20% of the augmented data is taken to be the test set. The test set is randomly chosen from the generated data pool and then separated from the training set. For every training iteration, the network sees the training data and tries to learn by adjusting the learnable parameters (see Section 2.1.4). Every 100th iteration (after the transfer learning Section 2.1.6), the current network state is tested and validated. This period is chosen, because the lower layer is already trained, and only the fully-connected layers at the end are fine-tuned. The evaluation and test step in the basic training occurs every 500th iteration.
As the test and training sets are from the same pool, the validation set should be from a completely different data pool. In addition, the validation accuracy should be the measurement taken for the comparison of the different networks. Therefore, real data is used in this approach.

Figure 4.11: Arial characters printed in different variations and photographed with different disorders.

As seen in Figure 4.11, multiple variations are printed to obtain a validation set with data from a real environment with real disorders. The photos are taken with perspective warps, reflections or shadows.

Figure 4.12: Characters extracted as 40x40 from the photographed validation papers from Figure 4.11.

To use the photographs in the training as a validation set, every character must be extracted and the size must be normalized. In Figure 4.12, all the characters are extracted to 40x40px, which results in a set size of 770 characters (approximately 22 images per character, though a few had to be removed because of low quality).

4.3 Network Architecture

As discussed in Section 1.1, the architecture is crucial to the outcome of the network. The requirement for the structure is to score results that are satisfying (>90%) and work for the user's use case, which also includes mobile phones. To guarantee that older phones can also handle the model, the architecture must be designed to respect it. Some different structures have already been introduced (Section 1.1). Most of them are performance costly and are too large (for example, some ResNets have up to hundreds of layers) to gain enough performance during a scan (detection and classification). Therefore, the LeNet-5 structure [LCJB+89], which can be seen in Figure 2.8, seems to provide a good fit with which to start. Also, the LeNet-5 was created with a limited scope with only

handwritten characters and is accordingly similar to this approach.

4.3.1 First approach - LeNet-5

As discussed above, the LeNet-5 has similar requirements with regard to what the structure was created for. In addition, the structure is flat, so the complexity of the LeNet-5 provides a good fit for our use case. The architecture (Figure 2.8) starts with the input of the network as a one-dimensional, 32x32-pixel image. The data is processed in one convolutional layer attached to a ReLu activation function (Figure 2.3). The output of the first convolutional layer is 28x28 pixels. This is because no padding (Section 2.2.1) is added to the original LeNet-5, which would result in a smaller output. The data is then downsized through a max-pooling layer (Section 2.2.2) to a 14x14-pixel resolution and processed in a second convolutional layer with an attached ReLu, which is also without padding, with an approximate output of 10x10-pixel resolution. Next, a max-pooling subsamples the input to a 5x5-pixel resolution and puts the result into the first fully-connected layer of 120 neurons. The output is passed through another ReLu activation function and passed on to the last fully-connected layer with 84 neurons, which then is passed into a Softmax (Section 2.1.3) function, which results in a result vector of 10 possible classes (0-9). Each convolutional layer has a number of biases which is similar to the number of filters in the layer. Thus, the first convolutional layer has six biases and the second one has 16.

For the training, the loss function cross-entropy [DBKMR05]—which is a popular loss function for classification—is used together with a regularizer. The regularizer is a value to watch during the scan. It takes all the loss functions from each layer and squares it, so if some weights become higher during the training, this parameter will also increase. This is not a good effect, because a weight is then relied upon too much. A few weights should not be much higher than the others, so the goal for the network is to decrease both values (regularizer and cross-entropy) simultaneously, with the priority on the cross-entropy.

Batch size	10,000
Iterations	75,000
Test check iteration	500
Validation check iteration	2,500
Validation set accuracy	47.28%
Best Validation set accuracy	50.25%
Training set accuracy	93.24%
Best Training set accuracy	93.24%

Table 4.1: Parameter and results from the training with the LeNet-5 implementation.

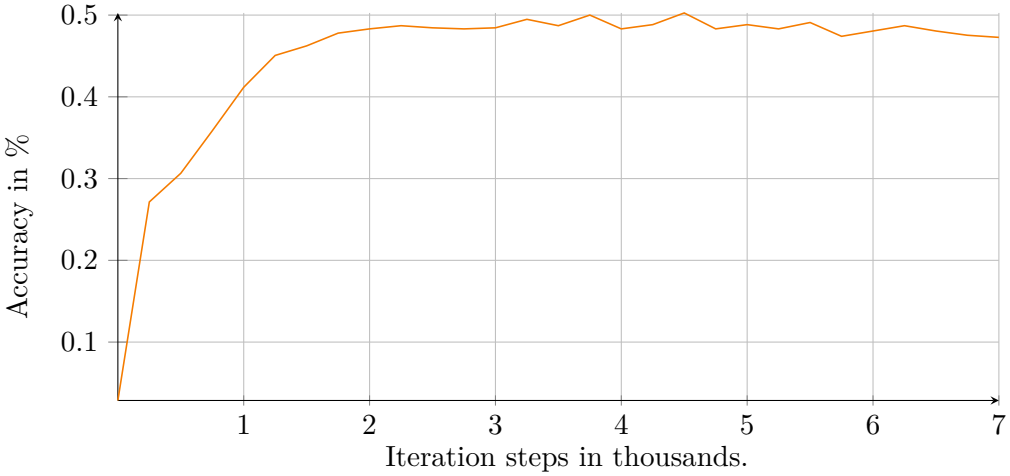


Figure 4.13: The validation set accuracy over 70,000 iterations.

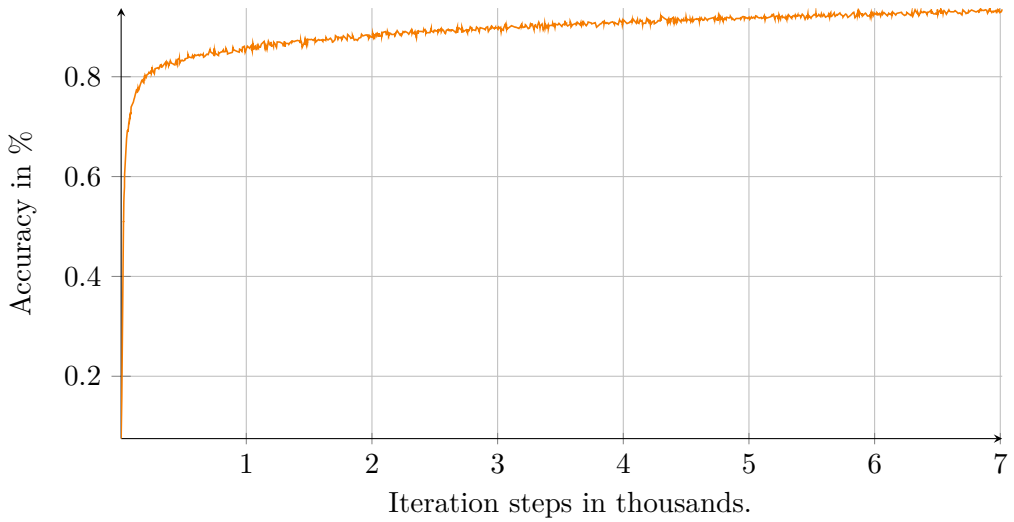


Figure 4.14: The training set accuracy over 70,000 iterations.

As shown in Table 4.1, the LeNet-5 approach does not score satisfying results. The training set accuracy is up to 93.24%, but the validation set accuracy is only 47.28%. This is a perfect example of overfitting. Too many iterations in the training lead to the result that the network becomes used to the training data. Nevertheless, the best accuracy obtained for the validation set is of 50.25%, which was taken at the 45,000th iteration, which is also not satisfying. Note this in Figure 4.13. There, the zenith of the validation accuracy is exceeded, and the graph decreases. In Figure 4.14, the graph is constantly increasing.

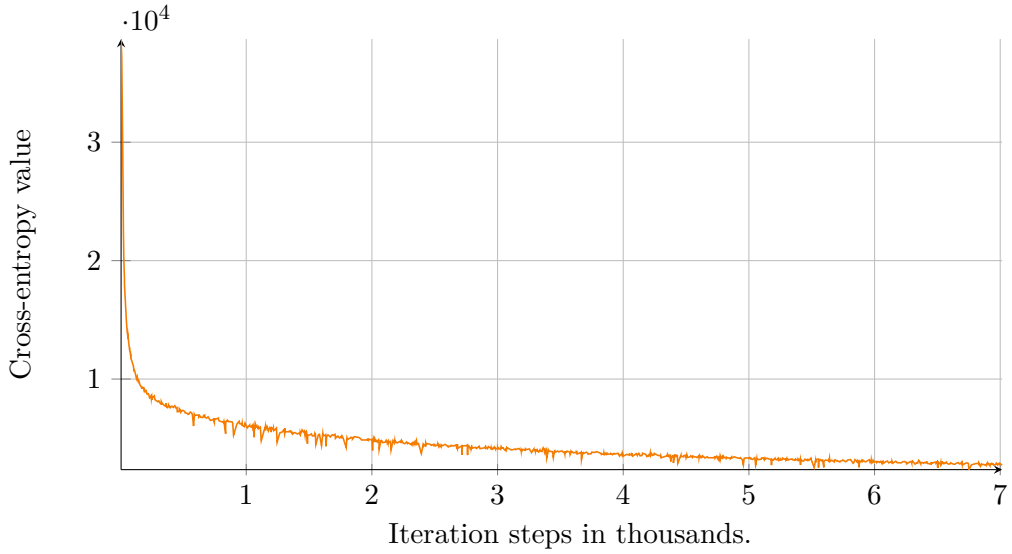


Figure 4.15: The cross-entropy loss over 70,000 iterations.

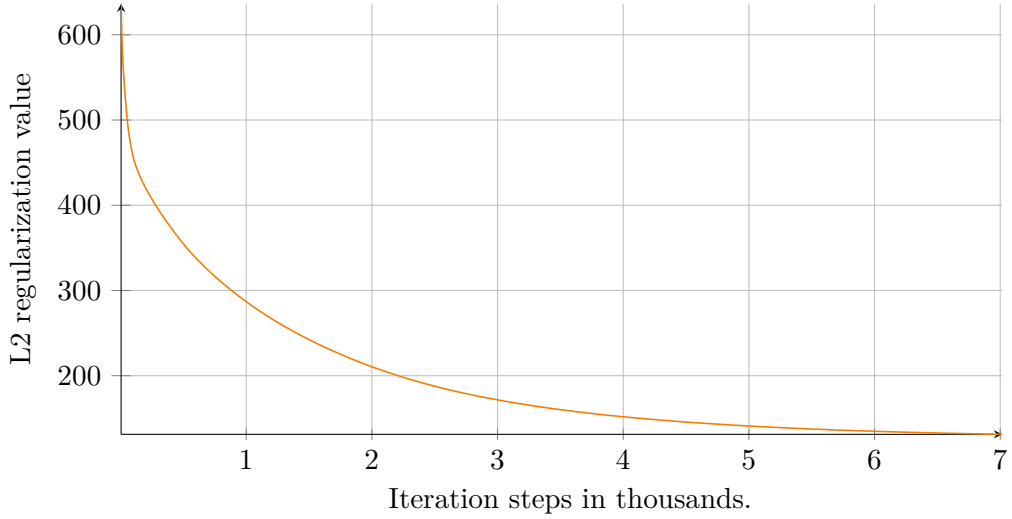


Figure 4.16: The regularization over 70,000 iterations.

Nevertheless, the cross-entropy loss function graph Figure 4.15 and the regularization graph Figure 4.16 are in a good shape, which is a asymptote tending to zero.
The problem with this is that the architecture is kept relatively small to detect more complex features. This is important for much disfigured examples. If the network has to classify an image that was taken under bad conditions—e.g., bad lightning, blurring

caused by movement or reflections—the network will not do a good job. Also, different fore- and back-grounds, which are inevitable in a real environment, are briefly covered by the approach. The downside in the LeNet-5 is that its input is black on white, which works well on the Modified National Institute of Standards and Technology (MNIST) dataset [LC10], but can lead to some problems in a real environment. The fore- or back-ground will not be pure white and black but will also have some other colours. This is why the image, before it can be processed by the network, must be pre-processed into a grayscale image. The colour is not going to add any classification information in the scope of character classification. This is of course different in other scopes: for example, general object classification in images. But for a character classification scope, only the amount of light in all the pixels is important.

4.3.2 Second approach - Adapting LeNet-5

The architecture of the network is shown in Figure 4.17. Here, an approach similar to that of the LeNet-5 is taken. What differs is the number of convolutional layers and the size of the inputs and outputs. At the beginning, an image with a higher resolution is chosen to retain more information. The input of the network is a one-dimensional, 40x40-pixel image. The exact process is described in Section 4.4. The number of biases for every layer corresponds exactly to the number of filters from the specific layer, as in the first approach. So, the first two convolutional layers have 16 and the last two layers have 32 biases. For the training, the Adam optimizer (Section 2.1.5) is used. Because the augmented dataset can be very large (> 1,000,000 examples), an SGD (Section 2.1.5) makes more sense. This helps to decrease the amount of data that is processed during the training.

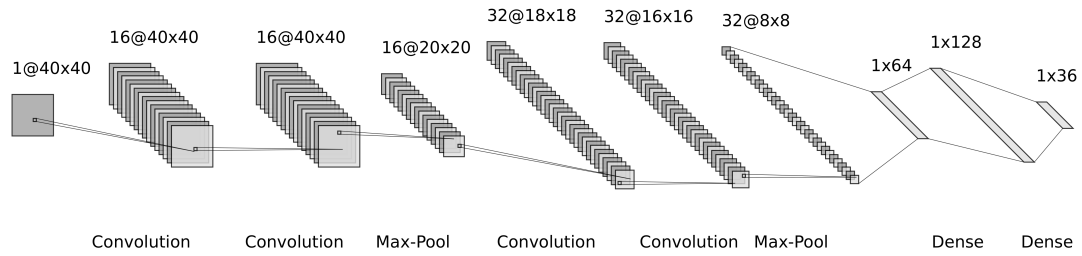


Figure 4.17: Architecture of the CNN for OCR with 36 classes (capitals and numbers).

4.4 Data processing

The image is processing some stages through all the layers in the network. Every layer has a unique

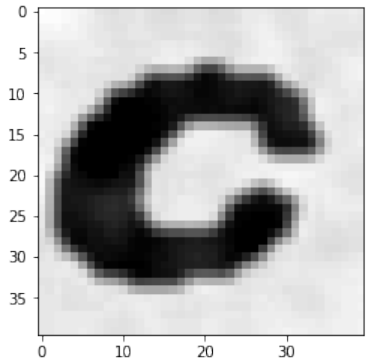
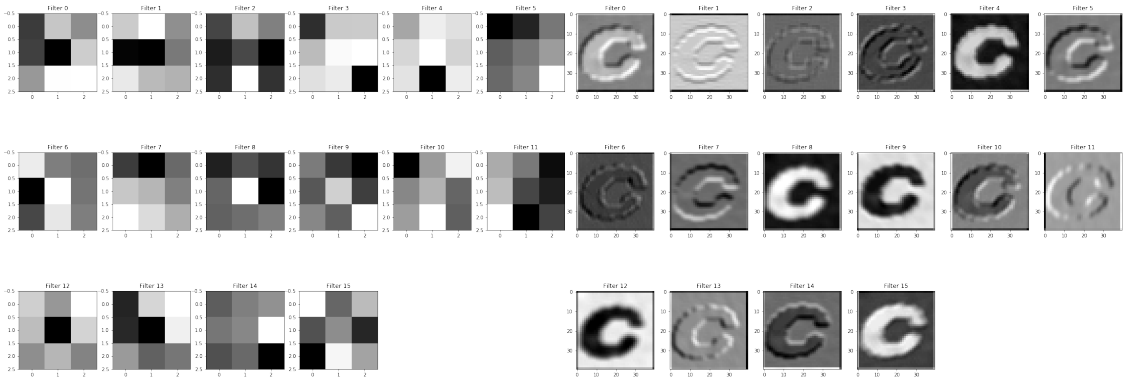


Figure 4.18: Input example of the network.

The input has only one dimension, because the color information is not necessary [LBBH98]. For OCR, the information is stored in spatial features; therefore the input image is grayscaled, seen in Figure 4.18.



(a) 16 3x3 filters. (b) 16 40x40 output.

Figure 4.19: Visualization of first convolutional layer filters and output before the activation function.

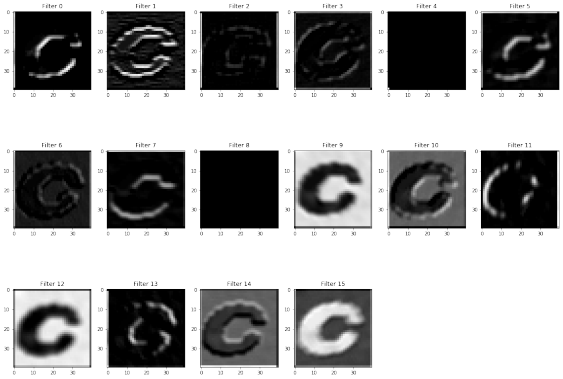
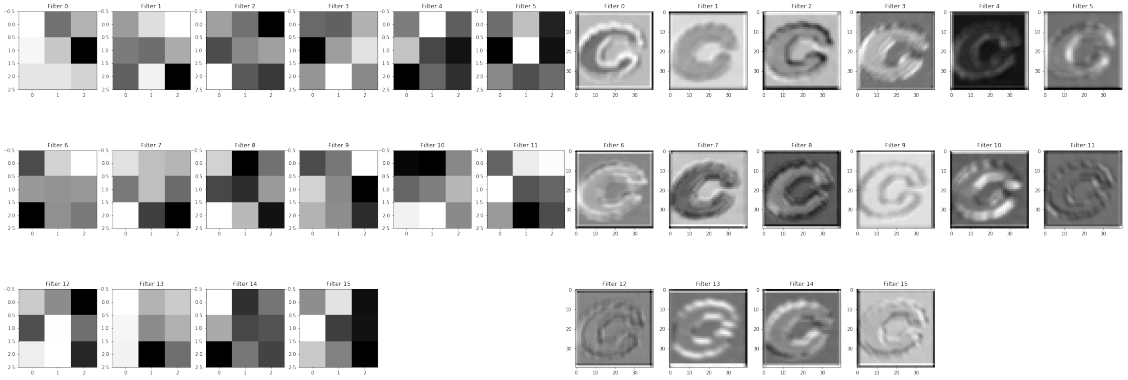


Figure 4.20: 16 40x40 output after ReLu.

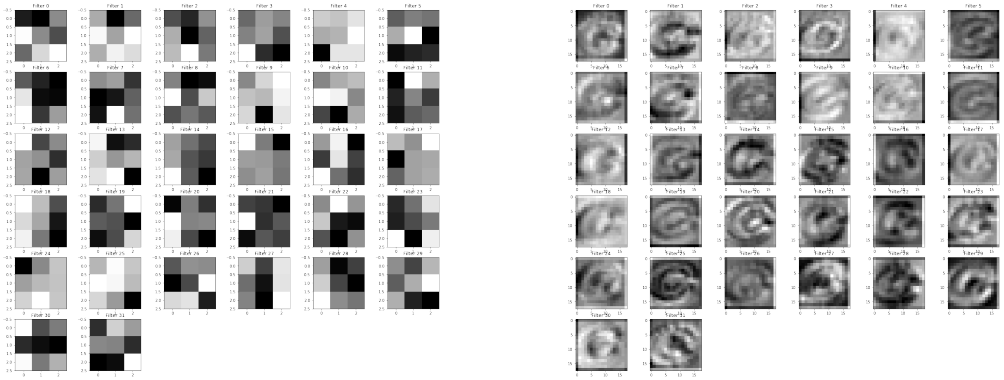
The first convolutional layer consists of 16 3x3 filters and has a ReLu for the output of the activation function. In Figure 4.19, a padding (Section 2.2.1) is chosen to make sure that all the information for the low-level features is not lost. In Figure 4.19a, the 16 trained filters are depicted. Each filter consists of nine weights (shown as a 3x3 matrix) which range from 0 - 255 (256 values), which results in 144 weights in the first layer. In Figure 4.19b, the output after the convolutional operation is shown before the ReLu, Finally, in Figure 4.20, the output after the ReLu is shown. The low-level features, such as edge detection, are clearly visible.



(a) 16 3x3 filters. (b) 16 40x40 output.

Figure 4.21: Visualization of second convolutional layer filters and output before the activation function.

The second convolutional layer (Figure 4.21) is similar to the first one with 16 3x3 filters, ReLu output and a zero-padding to remain the output size. This similarity of the first and second layers is used to cover all low-level patterns. A 2x2 max-pooling is applied to the output of the second convolutional layer, which downsized the input for the third convolutional layer from 40x40 to 20x20.

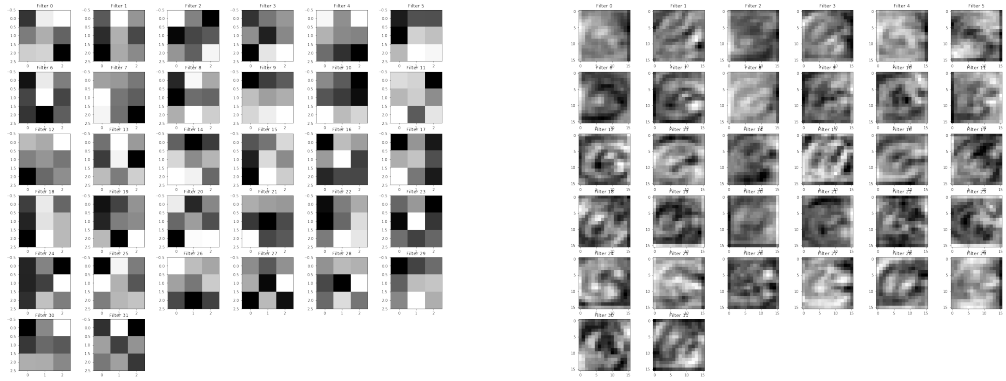


(a) 32 3x3 filters. (b) 32 18x18 output.

Figure 4.22: Visualization of third convolutional layer filters and output before the activation function.

As shown in Figure 4.22, the number of filters is doubled in the third convolutional layer because, after the max-pooling, different higher-level features are to be extracted

from the input. The layer consists of 32 3x3 filters. The convolutional operation here is without any padding, which will result in a decrease of the spatial size of the output (input is 20x20, output is 18x18). Also, in Figure 4.22b, the output of those smaller inputs will detect patterns specific to the class.



(a) 32 3x3 filters.

(b) 32 16x16 output.

Figure 4.23: Visualization of fourth convolutional layer filters and output before the activation function.

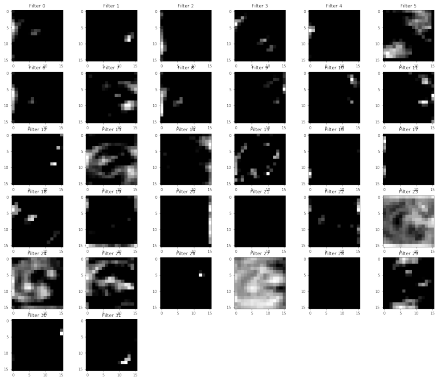


Figure 4.24: 32 16x16 output after ReLu.

The fourth layer (Figure 4.23) of the network is similar to the third layer and consists of 32 3x3 filters. The output of this layer after the ReLu activation function is to go into another max-pooling layer to downsize the input of 32 16x16 to 32 8x8, as shown in Figure 4.17. The max-pooled output is the input for the first fully-connected layer. At this state, the fully-connected layer looks at the output and determines which extracted

high-level features are most correlate to a particular class. The size of the first and second fully-connected layers are doubled by the size of the previous one. This approach is used to give every high-level feature multiple weights with which to classify the specific class. The last fully-connected layer is the size of the designated output. This example has a size of 36 because the network should classify 36 classes (26 capital letters and 10 digits). The output of this layer is to be used in a Softmax function, which is then the prediction of the network.

4.5 Pre-Trained Model

To provide a robust model for classification and fast training, the first four convolutional layers are pre-trained. The weights of this pre-trained model are to be loaded for the four layers, and the three fully-connected layers at the end of the network are to be retrained for the specific font. The data augmentation is different for the pre-training and the training. As in the pre-training, the lower convolutional layers are trained, which means that it is important to have a lot of variance in the training set. In addition, different fonts must be used to show the network that classes can differ. In this approach, 50 different fonts are used for the training. To create a proof of concept, only capitals and digits were trained, because the data augmentation would have taken double the time it took for half of the characters (exact times in Section 6.2)) to investigate the network architecture and hyperparameter of the data augmentation. In addition, in this prototype, only fonts without serifs are trained.

CHAPTER 5

Implementation

5.1 Hardware

Two Nvidia Titan Xs were used in the setup so we could benefit from faster training times. In addition, the VRAM of the graphic cards speed up the batch loading of the data. Larger batches can be loaded into the memory and processed. The work station has 64GB Ram and an Intel Core i7-6850K with 3.60GHz x 12. The CPU is very important to the data augmentation. There, a lot of image processing takes place, which runs in parallel.

5.2 Software and Frameworks

For the approach taken in this thesis, different software and frameworks are used to simplify the work. In addition, the applied libraries are state-of-the-art, which can guarantee a more solid code base.

- **Python** is a programming language. In this thesis, it is chosen for the main language because different concepts are implemented fast.
- **Tensorflow** is an open-source math library from Google [ABC+16] which is often used in ML applications. With the help Tensorflow, you can easily implement concepts like network structures or training processes.
- **Flask** is a Python web-framework.
- **Tensorboard** is an extension to Tensorflow, which reads and visualizes, amongst other things, event logs created by the programmer in Tensoflow .
- **Jupyter Notebook** is an interactive web-based environment for Python. Here, Python code can be outsourced in code cells and documented with text cells using markdown.
- **NumPy** is a Python library which adds, amongst other things, a better handling of arrays and matrices.
- **OpenCV** is a library for different computer vision tasks written in C/C++.

5.3 Structure

The structure of the application is simple.

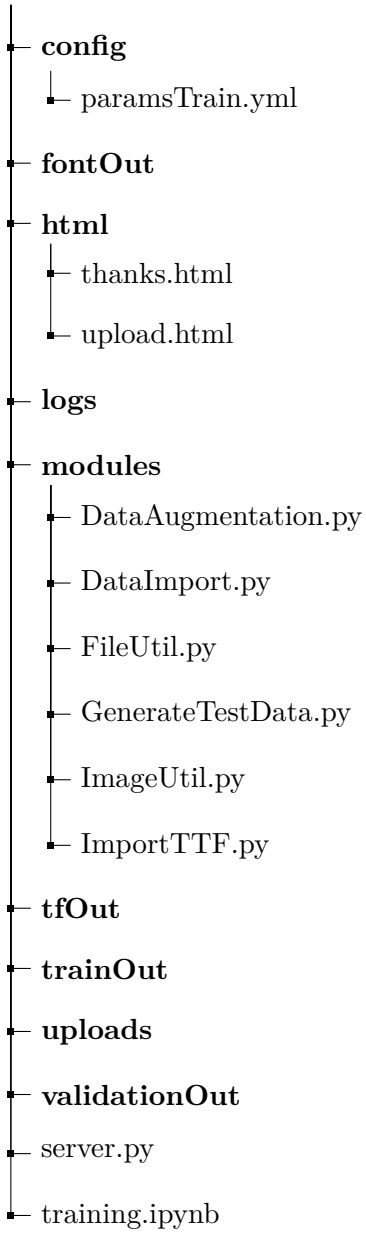


Figure 5.1: The Structure of the application. The bold written entries are directories.

5.4 Scripts

In this section, all of the different scripts and data flows are explained. The scripts for generating the data, creating the server and training the network are written in Python. With the help of Jupyter Notebook, the script in which the training of the model takes place can be categorized in different cells to keep a clean structure and environment. In addition, the adaption of different concepts is easier because you can rerun code cells and the Python kernel will stay in the same state, so all variables are already initialized. If split of the code is meaningful, the rerun will decrease the development time intensively.

5.4.1 Server

server.py

This file generates a small Flask server. The server has one root route, "/", where a GET and a POST request can be made. On the GET request, the server serves the upload.html, where the user can upload a font file. The POST request, which contains a **.ttf* file on the same route, will save the corresponding file to the upload folder for further usage. After a validation of the file extension, subsequent services are called.

1. Calling *ImportTTF.py* to export every character from the font file as *.png image as ground truth for the data augmentation. The exported images are saved in folders named after their corresponding ASCII code.

2. Calling *GenerateTestData.py* with the Augmentation file *DataAugmentation.py* including the configuration file *paramsTrain.yaml* for managing the data augmentation process.

3. Calling *Training.py*, which is the converted Jupyter Notebook. Here, the training is done.

5.4.2 Data Augmentation

As described above, the data augmentation consists of the extraction of the ground truth from the ttf file into single images in *ImportTTF.py* and the generation of all needed training and test data in DataAugmentation.py.

ImportTTF.py

This file is called first in the augmentation workflow to extract the characters as *pngs*. Besides the paths of the input and the output, a regular expression can be passed into the function *importFont*. The function attempts to extract every character that is valid from the regular expression into the desired output folder categorized in folders that are named after the ASCII of the extracted character. In addition, the colours of the image are adjusted so that ground truth is always white character with black background.

GenerateTestData.py

The generation of the training and test data is delegated by this file. It is called after the extraction of the ground truth. It iterates over every categorized ASCII folder in the given input directory and creates a pool of processes to iterate of every character augmentation simultaneously. Those processes receive the DataAugmentation.py file and its parameters, which consist of the grayscale image itself, a file stream, the label of the character (which is the ASCII code), and the destined size. To increase the performance of the training, all generated files and their labels are written into a stream. The idea is taken from the MNIST dataset, where the IDX file format is taken. The IDX file format is a simple format for vectors and multidimensional matrices of various numerical types [LC10]. If the millions of training images are saved as, for example, *pngs* to the hard drive, this would increase the size of the data immensely. In addition, the training process would waste too much time reading such a large number of single files. To avoid this, every single image is written into the stream and the label is also written to a separate stream. After the augmentation is finished, the streams are closed, compressed with gzip and renamed and categorized as *data_train<ASCII_Code>-<count>.gz* and

data_labels<ASCII_Code>-<count>.gz. The count is the number of the file with the same ASCII when a limit is set per compressed file.

DataAugmentation.py

This file is called in parallel from GenerateTestData.py. Here, the generation and manipulation are called from the ImageUtil.py. ImageUtil.py receives the augmentation configuration from the paramsTrain.yml file (Section 5.4.2). The manipulations that take place are discussed in Section 4.1. The whole process consists of nested loops for every kind of manipulation. At the end of the process, the image and the associated label are written to the streams.

Configuration file

For easier evaluation and development, the parameter of the data augmentation is outsourced. The **.yaml* file contains every configuration that can be taken for the augmentation process, as explained in Section 4.1.

- **numSteps**: The number of iteration steps of every algorithm (e.g., the rotation or translation).
- **dAngle**: The maximum angle of the rotation in the positive and negative directions.
- **dPerspX**: The number of steps for the perspective transformation on the X-Axis.
- **dPerspY**: The number of steps for the perspective transformation on the Y-Axis.
- **minPadding**: The minimum padding around the image. This is also used for scaling, because the output image always has the same size. Thus, with a higher padding, the image is smaller and vice versa.
- **maxPadding** The maximum padding around the image.
- **numBackgroundVals** How many different backgrounds are randomly chosen.

- **minBackground** The minimum value of the randomly chosen background range.
- **maxBackground** The maximum value of the randomly chosen background range.
- **minContrast** The minimum value of contrast that is added to the image.
- **maxContrast** The maximum value of contrast that is added to the image.
- **numForegroundVals** How many different foregrounds are randomly chosen.
- **kSizeBlurVals** The size of the gaussian kernel.
- **sigmaNoiseVals** The size of the gaussian kernel.
- **numTranslations** The steps of translations between 1 and -1.
- **reflectionSeed** The random seed of the reflection on the image.
- **pRefVals** The Reflection probability.

5.4.3 Utils

Many functions are used in multiple files or places. To avoid redundancy, util files are implemented.

FileUtil.py

This util file is for every small process that is needed to work with files such as writing images to the IDX file with all associated bytes or showing images from those files. Several other small functions are implemented here.

ImageUtil.py

This util file is for outsourced augmentation functions such as adding reflections, noise, or translations. Also, small helper functions like resizing the image or grayscaling it.

DataImport.py

This util file is for the actual training. It contains a class *DataSet* with multiple parameters and functions. The main functionality of this file is to work with the data sets, load them into the memory and feed the network with the right data for example with batches.

5.5 Training

In the development, a Jupyter Notebook **.ipynb* was used. This program provides a clean environment, as the technology is based on python cells which can be categorized and commented with markdown cells. In production mode, the python cells are combined into one python **.py* file, which can be used in the workflow.

5.5.1 Training parameters

- **Iterations**: The number of loops a training will take.
- **BatchSize**: The number of images that are used for one iteration in the training.
- **Test check iteration**: With each X iteration, the model tries to classify the test set, and the accuracy of that try will be measured.
- **Validation check iteration**: With each X iteration, the model tries to classify the validation set, and the accuracy of that try will be measured.

5.5.2 Architecture

The structure of the architecture is discussed in Section 4.3.2. The implementation is split into different situations. At first, the initialization functions of the weights and biases are defined. As seen in Listing 5.1, the weights are initialized randomly with the helper function of tensorflow *truncated_normal*. The standard deviation *stddev* is set to 0.1 at the weights to keep them as small as possible, as discussed in Section 2.1.4. The biases are initialized at 0.
Also, the function *conv2d* is defined, which is a small helper function used to create

a convolutional layer by the sizes from the parameters. The first parameter x is the dimension of the input and the second one W is the dimension of the weights for this specific layer. Padding can also be added. The default padding is *SAME*, which means that padding is added so that, after the convolutional operation, the output has the same size as the size of the input. The strides are hard-coded to 1 in each dimension, so the filter will not skip pixels.

```
1    def weight_variable(shape, name):
2        initial = tf.truncated_normal(shape, stddev=0.1)
3        return tf.Variable(initial, name=name)
4
5    def bias_variable(shape, name):
6        initial = tf.constant(0.0, shape=shape)
7        return tf.Variable(initial, name=name)
8
9    def conv2d(x, W, p='SAME'):
10       return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding=p)
```

Listing 5.1: Definition of the initialization from weights, biases and layers

Training of the convolutional layer

In Listing 5.2, we see the definition of the first convolutional layer of the network. At first, the size of the input and the sizes of the filters used in the first layer are defined. To get the size of the input, the input array *inputNodes* from the input image is reshaped, so the convolutional layer knows the dimensions. The first value is for batch normalization, which is not needed, so it is set to -1. Then height and weight are implemented. The fourth value of the image variable are the channels. Here it is 1, because the input is a grayscale image. After calling the initialization functions for the weights and biases, they are used in the initialization of the layer. The last step is the ReLu, which is added onto the output of the first layer.
In the second layer, instead of the input image, the output of the first layer is taken. This will continue through the whole architecture.

```
# INPUT LAYER
szInputX = 40
szInputY = 40

# FIRST CONVOLUTIONAL LAYER
szFilt1 = 3
nFilt1 = 16

with tf.name_scope("conv1"):

    # RESHAPED INPUT IMAGE
    image = tf.reshape(inputNodes, [-1, szInputY, szInputX, 1])

    # FIRST CONVOLUTIONAL LAYER

    W_conv1_class = weight_variable([szFilt1, szFilt1, 1, nFilt1], 'W_conv1_class')
    b_conv1_class = bias_variable([nFilt1], 'b_conv1_class')

    layer1 = conv2d(image, W_conv1_class, p='SAME') + b_conv1_class
    h_conv1_class = tf.nn.relu(layer1)
...
with tf.name_scope("conv2"):
    ...
    layer2 = conv2d(h_conv1_class, W_conv2_class, p='SAME') + b_conv2_class
```

Listing 5.2: Creating the first convolutional Layer for training

Loading the trained weights of the convolutional layers

The first convolutional layer was created for the training use case. But in our production application, we are using transfer learning (Section 2.1.6) to speed up the process. Instead of initializing the weights randomly, they are loaded from the most successful training approach.

In Listing 5.3, the variable weights are loaded via numpy from an *npz* file and then passed on to the initialization of weights and biases. This is a file format from numpy, which is used to save several arrays into a file because, if the training scores satisfying results, the weights and biases are exported. These are just large arrays, so they can be saved in an *npz* file. In addition, the weights and biases are constants, so they will not change during the training (in the transfer learning use case).

```
1    weights = np.load('/path/to/the/weights.npz')
2    ...
3    with tf.name_scope("conv1"):
4    # FIRST CONVOLUTIONAL LAYER
5        W_conv1_class = tf.constant(np.array(weights['W_conv1_class']).
     astype(np.float32))
6        b_conv1_class = tf.constant(np.array(weights['b_conv1_class']).
     astype(np.float32))
```

Listing 5.3: Creating the first convolutional Layer as constants and load weights

5.5.3 Evaluation and training

For the evaluation during the training, the cross-entropy loss function and regularization are used. As explained in Section 4.3.1, those values are used for the training, so the network tries to decrease them. In Listing 5.4, their declarations are shown. For the regularization, the outputs of the fully-connected layers are used. For the training, the Adam optimizer is used.

For the evaluation of the accuracy of the test and validation set during the training, a scope is declared which will cast the input into a float and take its mean.

```
1    # CROSS-ENTROPY + L2 REGULARIZATION
2    with tf.name_scope("cross_entropy"):
3        ...
4        regularizers = (tf.nn.l2_loss(W_fc1_class) + tf.nn.l2_loss(
     W_fc2_class) + tf.nn.l2_loss(W_fc3_class))
5        cross_entropy = tf.reduce_sum(tf.nn.
     softmax_cross_entropy_with_logits(logits=logits, labels=gtNodes)
```

```
                                    + 0.5*1e-3 * regularizers)
    ...
    # TRAIN
    with tf.name_scope("train"):
        train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
    # ACCURACY
    with tf.name_scope("accuracy"):
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Listing 5.4: Creating the evaluation of the training

In Listing 5.5, the training loop is shown. This loop goes on until the predefined *iterations* are reached. If the *itCounter* hits a number at which the modulus of the predefined *evalStep* or *validationEvalStep* is 0, the loop will jump into the condition. The previously defined accuracy *name_scope* is used for the validation and test-set accuracy check. Both are fed with the respective batch and the accuracy is printed.
The training uses the predefined *name_scopes* but will trigger every iteration.

```
    itCounter = 0
    ...
    for i in range(itCounter, itCounter + iterations):
        batch = trainSet.next_train_batch(batchSize, sampleLoadSize)
        summary1, _ = sess.run([stats_merged, train_step],
                        feed_dict={inputNodes: batch[0].reshape(-1,
    szInputX*szInputY),
                            gtNodes: batch[1],
                            whiteListNodes: np.ones(batch[1].shape,
                            dtype="f4")})
        ...
        itCounter += 1
```

Listing 5.5: The training block in the training loop

In the training loop, there are validation steps to check during the training, if the network is overfitting and how it is performing on data, it has not seen yet. A batch from the validation set is loaded and checked with the current network, as seen in **??** 5.6.

```
batchNoisy = validationSet.next_test_batch(batchSize, sampleLoadSize)
noisy_accuracy = sess.run(accuracy,
                    feed_dict={inputNodes: batchNoisy[0].reshape(-1,
szInputX*szInputY),
                        gtNodes: batchNoisy[1],
                        whiteListNodes: np.ones(batchNoisy[1].shape,
dtype="f4")})
```

Listing 5.6: The validation block in the training loop

In Listing 5.7 is the output of a training loop. This is the actual output that is used for the validation of the network and how it is performing.

```
will perform 2000 iterations.
step 0, training accuracy 0.0262
Test set is done...
------------------------
step 0, Validation set accuracy: 0.0350649
------------------------
step 300, training accuracy 0.9745
Test set is done...
------------------------
step 300, Validation set accuracy: 0.972727
------------------------
step 600, training accuracy 0.9892
Test set is done...
------------------------
...
```

Listing 5.7: The oputput of the training loop

CHAPTER 6

Results and Evaluation

As discussed in Section 2.1.4, the goal of a training is to find the minimum of the loss function (Section 2.1.4). As the initialization of the parameter is random, the minimum found by training is usually a local minimum. Therefore, it is common to repeat training with the same parameters to find the best minimum of all runs. This is why the results are shown in multiple runs.

6.1 Evaluation approach

For the evaluation of the different setups, the accuracy is measured for every run on the test and validation set described in Section 4.2. The test set is different for every augmentation run. For comparison between the different augmentation and training approaches, the accuracy of the validation set is crucial, as this set was generated once and used for the calculation of the accuracy by every approach. The accuracy is calculated as follows:

$$accuracy = \frac{TP}{TP + FP} \tag{6.1}$$

As seen in Equation (6.1), all true positives (True Positives (TPs)) are divided by all samples which consist of TPs and false positives (False Positives (FPs)), where TPs are all correctly classified examples from the respective set and FPs all incorrectly classified ones.
Because the initial aim, as expressed in Section 1.2, was to create an automated machine-learning system, the duration of a run can be crucial in combination with the accuracy.

6.2 Results

This section will show all the results from different approaches to train a model for a use-case. All the parameters from the data augmentation are explained in Section 5.4.2. The parameters from the training are described Section 5.5.1.
All subsections show the results of one generated set. All data from those subsections is generated differently. The properties and results of the corresponding training from the model with those sets are also documented.

6.2.1 First set

The first set exhibits a small level of mutation. The amount of data generated is, together with the second set, the highest. The generated data is to be trained in two different training scenarios. The first will have a smaller iteration count than the second one. In addition, the checks of the test and validation set will be further apart.

Parameter	Value
Rotation angle	$-5°$ to $5°$
Vertical perspective warp	16 steps
Horizontal perspective warp	16 steps
Min Padding	4
Max Padding	10
Blur Kernel	[3, 5]
Noise	[0.02, 0.035, 0.04, 0.045]
Translation steps between -1 and 1	2
Iterations of augmentation	2

Table 6.1: Parameter of the data augmentation

Generated images	9,289,728
Time to generate images	02h 02min 20sec

Table 6.2: Properties of the generated set

Parameter	Value
Batch size	10,000
Iterations	500
Test check iteration	100
Validation check iteration	100

Table 6.3: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	98.19%	757	13	98.31%	03min 34sec
Second run	98.12%	754	16	97.92%	03min 29sec
Third run	98.22%	748	22	97.14%	03min 31sec

Table 6.4: Results of the first data set containing 9,289,728 images and a training iteration of 500

Parameter	Value
Batch size	10,000
Iterations	2,000
Test check iteration	300
Validation check iteration	300

Table 6.5: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	99.65%	753	17	97.79%	12min 10sec
Second run	99.75%	749	21	97.27%	12min 21sec
Third run	99.6%	744	26	96.62%	12min 08sec

Table 6.6: Results of the first data set containing 9,289,728 images and a training iteration of 2,000

6.2.2 Second set

The second set has the same amount of generated images. The level of mutation in the augmented data is increased, so the test and training data will be more manipulated.

Parameter	Value
Rotation angle	$-8°$ to $8°$
Vertical perspective warp	20 steps
Horizontal perspective warp	20 steps
Min Padding	4
Max Padding	10
Blur Kernel	[3, 5]
Noise	[0.02, 0.035, 0.04, 0.045]
Translation steps between -1 and 1	2
Iterations of augmentation	2

Table 6.7: Parameter of the data augmentation

Generated images	9,289,728
Time to generate images	01h 59min 43sec

Table 6.8: Properties of the generated set

Parameter	Value
Batch size	1,000
Iterations	2,000
Test check iteration	100
Validation check iteration	100

Table 6.9: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	99.2%	745	25	96.75%	12min 15sec
Second run	99.6%	747	23	97.01%	12min 21sec

Table 6.10: Results of a data set containing 9,289,728 images and a training iteration of 2,000

6.2.3 Third set

In this set, the amount of data is decreased by fewer parameters and a small iteration of the augmentation. The level of manipulation is mostly the same as in the first set but with fewer noise variations and more translation steps. Thus, this set will be less disturbed but more translated.

Parameter	Value
Rotation angle	$-5°$ to $5°$
Vertical perspective warp	16 steps
Horizontal perspective warp	16 steps
Min Padding	4
Max Padding	10
Blur Kernel	[3, 5]
Noise	[0.02, 0.03]
Translation steps between -1 and 1	5
Iterations of augmentation	1

Table 6.11: Parameter of the data augmentation

Generated images	1,814,400
Time to generate images	28min 25sec

Table 6.12: Properties of the generated set

Parameter	Value
Batch size	10,000
Iterations	2,000
Test check iteration	100
Validation check iteration	100

Table 6.13: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	92.96%	672	98	87.27%	11min 58sec
Second run	93.22%	655	115	85.06%	12min 11sec

Table 6.14: Results of a data set containing 1,814,400 images and a training iteration of 2,000

6.2.4 Fourth set

This set is mostly the same as the first one, but with fewer images generated and less noise added. In addition, the two different trainings are done to yield a comparison between the first set and this set.

Parameter	Value
Rotation angle	$-5°$ to $5°$
Vertical perspective warp	16 steps
Horizontal perspective warp	16 steps
Min Padding	4
Max Padding	10
Blur Kernel	[3, 5]
Noise	[0.02, 0.04]
Translation steps between -1 and 1	2
Iterations of augmentation	1

Table 6.15: Parameter of the data augmentation

Generated images	1,814,400
Time to generate images	31min 02sec

Table 6.16: Properties of the generated set

Parameter	Value
Batch size	10,000
Iterations	500
Test check iteration	100
Validation check iteration	100

Table 6.17: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	99.95%	682	88	88.57%	12min 12sec
Second run	99.98%	694	76	90.12%	12min 16sec
Third run	99.92%	714	56	92.72%	12min 34sec

Table 6.18: Results of a data set containing 1,814,400 images and a training iteration of 500

Parameter	Value
Batch size	10,000
Iterations	2,000
Test check iteration	100
Validation check iteration	100

Table 6.19: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	99.98%	723	47	93.86%	12min 14sec
Second run	100%	702	68	91.16%	12min 31sec
Third run	99.99%	739	31	95.97%	12min 08sec

Table 6.20: Results of a data set containing 1,814,400 images and a training iteration of 500

6.2.5 Fifth set

The fifth and last set is generated with the least data. Also, the perspective is warped with a higher number of steps, so more images are warped.

Parameter	Value
Rotation angle	$-5°$ to $5°$
Vertical perspective warp	20 steps
Horizontal perspective warp	20 steps
Min Padding	4
Max Padding	10
Blur Kernel	[3, 5]
Noise	[0.02, 0.2, 0.5]
Translation steps between -1 and 1	2
Iterations of augmentation	1

Table 6.21: Parameter of the data augmentation

Generated images	435,456
Time to generate images	07min 30sec

Table 6.22: Properties of the generated set

Parameter	Value
Batch size	10,000
Iterations	2,000
Test check iteration	100
Validation check iteration	100

Table 6.23: Parameter of the training

Runs	Test set accuracy	Validation true	Validation false	Validation set accuracy	Duration
First run	83.96%	675	95	87.66%	12min 08sec
Second run	83.14%	679	91	88.18%	12min 21sec
Third run	83.88%	679	91	88.18%	12min 13sec

Table 6.24: Results of a data set containing 435,456 images and a training iteration of 2,000

6.3 Discussion of results

As described in Section 1.2, the problem this thesis is facing is to provide an automate system to train a machine learning model for a individual use-case.

Sets	Accuracy	Training set size	Training iterations	Augmentation duration	Training duration
First set	98.31%	9,289,728	500	02h 02min 20sec	03min 34sec
Second set	97.01%	9,289,728	2,000	01h 59min 43sec	13min 51sec
Third set	87.27%	1,814,400	2,000	28min 25sec	11min 58sec
Fourth set	95.97%	1,814,400	2,000	31min 02sec	12min 08sec
Fifth set	88.18%	435,456	2,000	07min 30sec	12min 13sec

Table 6.25: Summary of the highest accuracy from each set from Section 6.2

By looking at the different results in Section 6.2, we can see that it is possible to score a high accuracy for the model in a real environment. In Section 6.2.1 and Section 6.2.2, the largest training sets are generated. Also, the highest accuracy for the validation set is scored by the first set.

The largest difference between the first two sets is in the intensity of the hyperparameter of the data augmentation. The angle of the rotation and the perspective warp are increased in the second augmentation script, which leads to the higher mutation of the characters. As seen on the validation set, the second set is slightly less accurate. This does not mean that the higher mutation leads to a worse augmentation, as this effect may be just another local minimum on the loss function.

In the third set from Section 6.2.3, the accuracy is the lowest of all. Here, the translation steps are increased, so the character's position in the image is shifted in more steps. The fourth set in Section 6.2.4 exhibits good accuracy but only with a higher number of iterations. The fifth set in Section 6.2.5 is less accurate, like the third set. Note that the number of created images is more critical than the level of mutation. In addition, the number of iterations really depends on the randomly instantiated filters, which describe the start point on the loss function. More iterations may only lead to overfitting (Section 2.1.4), which can be seen on the fourth set in the second run with 2,000 iterations. There, the model classifies 100% on the test set, but it exhibits decreased accuracy on

the validation set.

Also, the use of a large number of iterations seems not to be a good choice. In most of the cases, most of the accuracy was achieved at the beginning of the training. In Table 6.26, we see that, after 300 iterations, the model scores 97.27% accuracy on the validation set. The best accuracy scored on the validation set is between 900 and 1,200 iterations. After that, only the accuracy of the training set increases but the accuracy on the validation set decreases. This is caused by overfitting the model.

Iteration	Training set accuracy	Validation set accuracy
0	2.83%	3.37%
300	97.02%	97.27%
600	98.87%	97.53%
900	99.47%	97.66%
1200	99.61%	97.66%
1500	99.63%	97.53%
1800	99.75%	97.27%

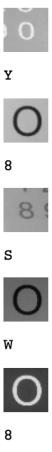
Table 6.26: Accuracy for every iteration during training from Section 6.2.2, second training and second run

The time that is used for the training is not very intense, because it is always under 15 minutes. The augmentation time is more crucial, because this can last for over two hours, as seen in the first two sets. Nevertheless, those sets exhibit the highest accuracy. But the sets with fewer images—or even the fifth set, which has the very fewest images—are still scoring 88.18% accuracy. Thus, for a development, it would be helpful to decrease the amount of data and generate a quickly generated model for a specific use case so as to implement it in an application for testing. For production, the amount of data can be increased to score more accuracy.

Depending on the detection, the level of mutation should not be high. For example, the difference between the third and the fourth sets is immense. Those sets are not very different, but the third set includes more translation steps. This will more cover a badly detected character, but it also decreases the accuracy. This is the same as with the perspective warp or rotation angle. Those parameters will increase the chance for the

model to classify images that are not well detected, but they also increase the chance of wrong classification.

(a) Wrongly classified examples of the fifth set from the first run.



(b) Wrongly classified examples of the second set from the first run.

(c) Wrongly classified examples of the first set from the first run and first training.

(d) Wrongly classified examples of the first set from the first run and second training.

Figure 6.1: Wrongly classified examples from two different sets. The character below the image shows the prediction of the model.

Here, the networks with the higher levels of mutation classify easy examples wrongly. The networks with the least level of mutation on the generated dataset are not making such mistakes and are failing only on the examples that are hard to classify. This can be seen in Figure 6.1. The network that had a training with a higher level of mutated data

predicts wrongly for the easily classifiable characters. In Figure 6.1a and Figure 6.1b, this behaviour can be seen. Also, the second set exhibits large troubles with the letter "O".

If we look at the false predictions from the first set in Figure 6.1c and Figure 6.1d, the examples are harder to classify, as they include a lot of distractions like reflection, perspective warp, translation or neighbouring characters that are detected in the image. In addition, the difference between the two trainings in set one is not very great. They fail both on mostly the same examples.

CHAPTER 7

Conclusion and future work

To summarize, I first discussed my motivation, problem, and the goal for the thesis. Next, I introduced ANN in general, beginning with the perceptron and leading over to CNN in general. Then I explained what our approach requires. Afterwards, I discussed similar and related work. Specifically, I discussed work in which neural networks are trained to classify characters or data is generated for deep learning training. In the next chapter, I discuss our methodology by explaining and visualizing how I will generate the data and structure the network. Next, the implementation of the approach is explained and documented with respect to each parameter that is important to this study. Finally, the results are documented for every different set that was generated which is relevant to the explanation (in the process, many more sets were generated) and discussed with regard to the relevance of the problem statement.

7.1 Conclusion

It was hard to reach the goal of keeping the structure simple and less complex but scoring a high accuracy in a real environment. Most related networks that classify images do not have a limit in the performance. To use this network in production, workstations with good hardware should be able to process. Devices like older mobile phones can use it. The network that is generated in this application can be production ready in less than an hour and score an accuracy greater than 90% in a real environment, depending on the use case. If we were to create a similar application without deep learning and neural networks by using different computer vision operations, the scope of the project would be greatly increased. Thus, neural networks are a good fit for cases like this.

Data augmentation in deep learning is very helpful in many use cases, because it is an easy method for getting enough labelled data with which to train an ANN. The results show that small and large data sets lead to a satisfying product. In addition, the smaller sets, which are generated more quicker than the large ones, deliver results that can be implemented in an industrial production environment, where time can also be a crucial factor. Nevertheless, the project is not perfect and can be adjusted and improved at many points. The possible combinations of the hyperparameters are countless and, at the end, most of the approaches are taken with the trial and error principal.

7.2 Future Work

To improve the speed of the whole system, the augmentation and the training can be ported to the GPU to make use of a multi-GPU concept. This, depending on the hardware, can improve the speed of the application.
For more variety or accuracy, different networks and pre-trained weights can be added to the system, so it could also classify another limited set of objects (except characters). This can also lead to new manipulations of the data sets because, for other objects, the colour for example, can be a critical factor for the classification (as discussed earlier, for OCR, greyscale images are enough).
At this point, the network is pre-trained to sans serif fonts. This can also be expanded to many more types of fonts. In addition, the implementation for the user to train multiple fonts at the same time and receive a network that can score good results on every use case would be meaningful.
Another interesting improvement could involve the setup of the server and the python kernels. Those could be clustered so the system can perform multiple trainings at a time.
At this point, the training is mostly done without a Graphical User Interface (GUI). To improve the user experience, more frontend elements could be added.

APPENDIX A

Repository

The repository is hosted by the university's GitLab server.
`https://git01lab.cs.univie.ac.at/a1369053/train_your_tensorflow`
Every information including the setup is provided in the ReadMe.md file, which is located in the root of the repository.

Figure A.1: The QR Code including the link to the repository.

List of Figures

List of Tables

Listings

Glossary

ASCII American Standard Code for Information Interchange. 28, 50–52

hyperparameter Hyperparameters are parameters in the scope of a machine learning training, that are set before the training starts.. xi, 2, 9, 16, 17, 69, 74

IoT The Internet of Things is a term to describe the connection or communication between devices embedded with software which creates a direct integration of those devices in computer-based systems.. 1

Tensorflow Tensorflow is an open-source math library from Google which is used for machine learning applications like neural networks.. xi

Acronyms

Adam Adaptive Moment Estimation. 17, 40, 57

ANN Artificial Neural Network. 5, 6, 15, 19, 73, 74

CNN Convolutional Neural Network. 3, 4, 8, 11, 12, 18, 19, 25, 26, 41, 73, 79, 80

FP False Positive. 62

GUI Graphical User Interface. 75

LSVRC Large Scale Visual Recognition Challenge. 25

ML Machine Learning. ix, xi, 4, 25, 48

MNIST Modified National Institute of Standards and Technology. 40

MSE Mean Square Error. 10

NN Neural Network. 4, 6, 7, 13, 16, 19, 79

OCR Optical Character Recognition. ix, xi, 3, 25, 26, 41, 75, 80

ReLu Rectified Linear Unit. 8, 25, 42–44, 55, 80, 81

RoI Region of Interest. 28, 29, 31

SGD Stochastic Gradient Descent. 16, 17, 40

TP True Positive. 62

Bibliography

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

[All01] Paul D Allison. *Missing data*, volume 136. Sage publications, 2001.

[And16a] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/assets/nn1/neuron_model.jpeg`, 2016. [Online; accessed April 08, 2018].

[And16b] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/assets/nn1/neural_net2.jpeg`, 2016. [Online; accessed April 08, 2018].

[And16c] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`, 2016. [Online; accessed April 15, 2018].

[DBKMR05] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[Dee06] Deep learning - moving beyond shallow machine learning since 2006! http://deeplearning.net/datasets/, 2006. [Online; accessed May 28, 2018].

[DSH13] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE, 2013.

[FCNL12] Clément Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Scene parsing with multiscale feature learning, purity trees, and optimal covers. *arXiv preprint arXiv:1202.2160*, 2012.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[GD04] Christophe Garcia and Manolis Delakis. Convolutional face finder: A neural architecture for fast and robust face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 26(11):1408–1423, 2004.

[HN92] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[HZC+17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

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

[KSK+16] D. G. Ko, S. H. Song, K. M. Kang, S. W. Han, and J. H. Yi. Optical character recognition performance comparison of convolutional neural networks and tesseract. *The 31st International Technical Conference on Circuits/Systems, Computers and Communications Technical Program*, 61:871–874, 2016.

[KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

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

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[LCJB+89] Yann Le Cun, LD Jackel, B Boser, JS Denker, HP Graf, Isabelle Guyon, Don Henderson, RE Howard, and W Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[LDK+15] Yisheng Lv, Yanjie Duan, Wenwen Kang, Zhengxi Li, and Fei-Yue Wang. Traffic flow prediction with big data: a deep learning approach. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):865–873, 2015.

[Nie18] Michael A. Nielsen. Neural networks and deep learning. `http://neuralnetworksanddeeplearning.com/`, 2018. [Online; accessed June 03 , 2018].

[PRH+14] Mattis Paulin, Jérôme Revaud, Zaid Harchaoui, Florent Perronnin, and Cordelia Schmid. Transformation pursuit for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 3646–3653. IEEE, 2014.

[PY10] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

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

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[RLF15] Artem Rozantsev, Vincent Lepetit, and Pascal Fua. On rendering synthetic images for training an object detector. *Computer Vision and Image Understanding*, 137:24–37, 2015.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review 65*, pages 386–408, 1958.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

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

[SG07] Zohra Saidane and Christophe Garcia. Automatic scene text recognition using a convolutional neural network. In *Workshop on Camera-Based Document Analysis and Recognition*, volume 1, 2007.

[SHK+14] 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.

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

[Smi07] Ray Smith. An overview of the tesseract ocr engine. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 2, pages 629–633. IEEE, 2007.

[WFHP16] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[YCBL14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.