Arbeitsgruppe für
Anwendungsspezifisch
e Multi-Core
Architekturen

RUB

OpenCL Deep Learning Framework for Automated Text Recognition

Master Thesis

Written by

Kappen, Patrick

30.06.2017

Prof. Dr.-Ing. Diana Göhringer
M. Sc. Lester Kalms

Abstract

Automated handwritten character recognition is a challenging task and can't easily be solved by using classical Algorithms. An alternative approach is Deep Learning, which has proven to be quite successful, but needs many computational resources. To solve those problems, an OpenCL deep learning framework was developed. Automated handwritten character recognition is solved by applying different Neural Networks. Normal encoder based Neural Networks suffer from the high dimensionality of images and the requirement of fixed image sizes. Thus, a contribution of this work is the use of sliding windows to increase the flexibility of automated handwritten character recognition by allowing flexible image sizes. The implemented Neural Networks, which used the sliding window approach, reached results comparable to the literature.

Content

0.1. Table of Contents

0.2. List of Figures

0.3. List of Tables

0.4. List of Equations

0.5. List of Code

1. Introduction

Humans create many handwritten documents in different areas and in some cases, it might be necessary to store this written text in a database or to make use of the information contained in such documents. One way to transfer this information from the physical world to the digital is to let humans read the document and then let them insert the information into a pc. Such work can be costly and tedious if it needs to be performed for many documents. An alternative automated solution is therefore necessary. Another way to solve this problem is to make use of algorithms that extract the data from images of the documents. A problem that arises when creating such algorithms is that reading handwritten text from images is not easy because the human writing underlies much variation. Also, the quality of the image and the illumination conditions may influence the appearance of the written text in the image. While the creation of a specific algorithm may be error prone an alternative solution is to turn to machine learning. Machine learning allows the system to learn from data and adapt itself to the task, making it more feasible. One area of machine learning that had proven to be quite successful recently is deep learning. Deep learning has proven to be a good approach to vision related problem, allowing the computer to extract information from many different images. They are loosely inspired by the human brain and its neurons and work by training an ensemble of abstract neuron objects, this ensemble is called a Neural Network. To train a Neural Network much data and computational resources are needed. While the available amount of data is task specific, GPUs have proven to be very useful in providing the necessary computational resources. Because of its recent success, many different deep learning frameworks have emerged, allowing the fast and easy creation of different Neural Network models. They also reduce the amount of knowledge a user needs to have, to apply deep learning on GPUs by abstracting away the details of the hardware. Through this development, a bias towards a specific heterogenous programming API called CUDA has emerged. While the implementations using CUDA are very efficient, they are also very restrictive because CUDA is only supported by NVIDIA GPUs. Another API allowing the use of GPUs is called OpenCL which is an open standard. It is supported by many different devices and not only by GPUs. A wide spread support of OpenCL would allow a direct application of deep learning on FPGAs and Embedded Systems. A focus of this work is therefore, to create a deep learning framework for handwritten character recognition using OpenCL. While it is important to decide on which frameworks and hardware to use, it is even more important to decide on the specific architecture of the Neural Network. There exist many different approaches on how such a problem can be

solved, ranging from an encoding of the image as start state for an RNN to more complex networks that support attention. This decision has major impact of the achievable performance and on the computational resources that are needed to apply such Neural Networks. This work will therefore analyze different Neural Network architectures which can be used to solve handwritten character recognition. This work was created in cooperation with the Opta Data group in Germany where the goal was to introduce Deep Learning and apply it to various character recognition related problems. This work will start by giving some background on Deep Learning and computational resources in chapter two. This is followed by a summary of relevant work in the field of handwritten character recognition in the context of deep learning and alternative deep learning frameworks in chapter three. Chapter four will describe the implemented framework and Neural Networks. The fifth chapter will state the results of this work followed by a conclusion and outlook in chapter six.

2. Preliminaries

In order to successfully apply deep learning to different applications, it needs to be understood, how deep learning works. Also, the computation of the different algorithms on the GPU is not trivial. This chapter will therefore give a basic introduction to different deep learning techniques needed, in the first part. The second part will then introduce GPU programming and the algorithms that are used to compute the operations necessary in deep learning.

2.1. Deep Learning

Deep Learning is a class of machine learning algorithms and consists of many different algorithms, which can be combined in many ways. This results in a wide range of applications one can try to solve with it, ranging from image classification to tasks like language translation and speech recognition. Deep learning solves tasks with models called Neural Networks which are loosely inspired by neurons in the human brain. They are made up of many layers of different neurons which are stacked over each other, therefore creating a somewhat deep system, hence the name Deep Learning. The Neural Networks used in deep learning can roughly be separated into two different classes. The first class are the Feed-Forward Neural Networks. They compute the output by using only the input and the parameters of this model:

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) \tag{2.1}$$

$\boldsymbol{y}$ represents the output of the Neural Network, $\boldsymbol{x}$ is the input to the Neural Network and $\boldsymbol{\theta}$ represent the parameters, of the Neural Network. The parameters are the part that is trained by a learning algorithm. They are very useful at tasks where the input is not sequential and has no time dependencies.

The second type of Neural Networks are the Recurrent Neural Networks. In contrast to the Feed-Forward Neural Networks, the output is not only computed using the direct input but also by taking information from a series of previous inputs. Therefore, they have a state which depends on the series of inputs. They compute not only one output but in general a whole series of outputs. Recurrent Neural Networks are very useful at tasks where the model needs to receive a sequence as input, needs to output one or both at the same time. They general output of a Recurrent Neural Network is computed in the following way:

$$\boldsymbol{y}_t, \mathbf{s}_t = f(\boldsymbol{x}_t, \mathbf{s}_{t-1}, \boldsymbol{\theta}) \tag{2.2}$$

Here $\boldsymbol{\theta}$ represents the parameters that are learned and $\mathbf{s}$ the state at the time step t, $t-1$ respectively.

While it is important what kind of model is used, the specific parameters of the model are even more important. They are the part of the model that is adapted to the specific task at hand and must therefore be learned using data samples of the task to be solved, also called data-sets. For this purpose, deep learning contains the Backpropagation algorithm for Feed-Forward Neural Networks and Backpropagation Though Time for Recurrent Neural Networks, for computing the gradients of the system and different optimization algorithms to update them.

This section will start by introducing Multilayer Perceptron's, the most basic type of Feed-Forward Neural Network. This is followed by a sub chapter concerning itself with Gradient Descent. Afterwards Backpropagation is explained and then Convolutional Neural Networks which are a type of Feed-Forward Neural Networks often applied to task where the input is an image. Basic Recurrent Neural Networks will then be explained followed by the Backpropagation Through Time algorithm. This then leads to different more advanced types of Recurrent Neural Networks, the LSTM and GRU cells, which will then be explained.

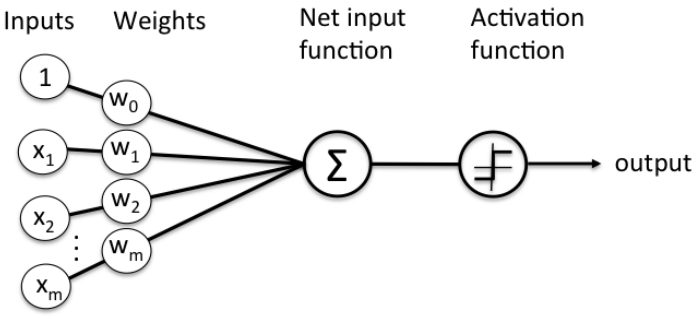

Figure 2.1 Single Perceptron [1]

2.1.1. Multilayer Perceptron

The Multilayer Perceptron is based on the work of [2] and a single Perceptron is displayed in Figure 2.1. The displayed perceptron calculates its output by calculating a weighted sum over all inputs and then applying a step function. The constant input one represents the so-called bias and is used to increase or decrease the output of the perceptron disregarding the input the step function is a so-called activation function. The weights of the sum are the parameters which are learned. In general, many perceptron's are used in parallel and form together a layer of a multilayer neural network. To be multilayer, the output of one layer is then the input into the next layer.

The computation of all weighted sums from one layer can be represented by a linear equation:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.3}$$

Here $\boldsymbol{W}$ represents a $M \times N$ matrix where N is the input dimension and M is the number of perceptron's in one layer. $\boldsymbol{x}$ is the input vector and is of dimension N. $\boldsymbol{b}$ represents the bias and has dimension M as does the Output $\boldsymbol{y}$. From this follows that if all activation functions, in every layer, are linear the total output of the network is linear in the input. For this reason, the activation function is often called non-linearity because it adds a non-linearity in the input data to the system also this way they are general function approximators [3]. There are many different non-linearities for example the sigmoid, tangents hyperbolics and the so called ReLU [4]:

$$max(0, x) \tag{2.4}$$

Parameters, that are not trained are called hyperparameters. Examples of hyperparameters are the activation function, the number of layers and the size of each layer. NNs are

especially powerful because they allow the extraction of features from training sets and allow the use of features adapted to the problem at hand, reducing the need to handcraft features that are used as input for the learning system [5]. This way the application of deep learning requires less knowledge of the target problem domain and more knowledge about the learning system making them very flexible for different applications.

2.1.2. Gradient Descent

Gradient Descent(GD) is an algorithm that can be used for finding a solution to problems of the following type:

$$min(f(\boldsymbol{x}, \boldsymbol{\theta})) \tag{2.5}$$

f is a cost function which should be minimized with respect to the parameters $\boldsymbol{\theta}$. $\boldsymbol{x}$ represents the input into the function. It was first described in detail by P.A. Nekrasov in year 1884 [6]. GD solves this problem by calculating the derivative of the cost function with respect to the parameters $\boldsymbol{\theta}$ and then changing it a bit in the opposite direction:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t) \tag{2.6}$$

Here α represents the learning rate and $\boldsymbol{\theta}_{t+1}, \boldsymbol{\theta}_t$ the parameters at step $t+1$, t respectively. In the context of deep learning with supervised learning one wants to minimize the so-called risk:

$$R(\boldsymbol{\theta}) = E[L(\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{y})] \tag{2.7}$$

L is a real valued loss function, $\boldsymbol{h}$ is the output of a function approximator given an input $\boldsymbol{x}$ and $\boldsymbol{y}$ is the respective output. Common examples for the loss are the mean squared loss $L_{MSE} = \sum_i (\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta})_i - \boldsymbol{y}_i)^2$ and the cross entropy $L_{CE} = \sum_i \boldsymbol{y}_i \log(\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta})_i)$, in this case $\boldsymbol{h}(\boldsymbol{x}, \boldsymbol{\theta})_i$ represents the probability of class i. In most cases, it is not possible to minimize the true risk because the true distribution of $\boldsymbol{y}$ given $\boldsymbol{x}$ is often not known because it is essentially the element that should be learned. One solution to this problem is to take samples of the true distribution, the training set, and then minimize the expected loss of the samples:

$$R_{emp} = \frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{h}(\boldsymbol{x}_i, \boldsymbol{\theta}), \boldsymbol{y}_i) \tag{2.8}$$

This is called empirical risk minimization because it approximates the true risk by taking empirical samples [7]. m represents the number of samples used for training.

There are three ways one might apply GD to minimize the empirical risk which all differ on the variance and computational cost. The first is the normal gradient descent where

the gradient is computed using all training examples as in equation ((2.8). This is in the context of deep learning often not practical because data sets often contain thousands to millions of examples which makes the full computation of the gradient prohibitively expensive for every update step.

The second and thirst approach make use of the so called Stochastic Gradient Descent (SGD) [8]. Each step SGD computes the gradient only for a few training examples.

Therefore, it only estimates the true gradient stochastically. This approach is by far more tractable than full gradient descent but it also suffers from high variance because the gradient estimate strongly depends on the sample [9]. It is also possible that SGD does not converge because it might oscillate around the minimum. One solution to this is to reduce the learning rate gradually [10]. In general, it takes more steps until convergence [11] but each step is by a large amount cheaper and it also allows full online learning because new examples can directly be used to improve the performance of the NN.

The third approach is a combination of the first two. It also computes a stochastic estimate of the real gradient but instead of taking only one sample it takes several examples which is called a mini-batch.

Mini-Batch GD does not suffer so much from high variance as does SGD. Also, it is more tractable to compute than the full batch GD

There exist a few problems with GD the first is that GD makes only use of the gradient which is a local feature of the cost function. It takes no global features of the function into account therefore there is no guarantee that GD will find a global optimum. An exception exists in the case of convex function where it is guaranteed to converge to a global optimum instead of a local optimum. A second problem exist when the cost function surface contains regions where the gradient is near zero or zero [12]. In the case of zero gradient GD might get completely stuck but also for near zero gradient the optimization might take very long because it takes a while for GD to escape those regions.

There exist a few improvements for GD which often allow faster convergence of GD. The first improvement is the introduction of a momentum term [13] which gives the following update rule:

$$\boldsymbol{v}_{t+1} = \mu \boldsymbol{v}_t - \alpha \nabla f(\boldsymbol{\theta}_t) \tag{2.9}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_{t+1} \tag{2.10}$$

Here the momentum $\boldsymbol{v}$ is computed by taking a running average of the gradients where the influence of the older gradients is exponentially decaying and μ is the decay factor. This results in a smoothing of the gradient over multiple time steps by decreasing

oscillations and increases the performance of the gradient in regions where the gradient in the wished-for direction is small or near zero. An alternative type of momentum is the Nesterov accelerated gradient [14]. The next improvement for GD is an adaptive learning. Here the goal is to change the influence of gradient updates based on the frequency of updates done to a parameter. If a parameter is only sparsely updated, the sparse updates should have a big impact in contrast to updates of parameters than are frequently updated. There are three algorithms that archive this behaviour, those are Adagrad [15], Adadelta [16] and RMSprop [17]. All three of them calculate the second momentum of the parameter gradients and then divide the learning by it.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\boldsymbol{G}_t + \varepsilon}} \odot \nabla f(\boldsymbol{\theta}_t) \tag{2.11}$$

Here ε is smoothing term that prevents division by zero, $\odot$ is the Hadamard product also called element wise product, $\boldsymbol{G}$ is the second momentum term. The main difference between those three algorithms is the way they calculate the second momentum term $\boldsymbol{G}$, it is for example computed by an average or running average. There exists also an algorithm that makes use of momentum and uses an adaptive learning rate. This algorithm is called Adaptive Moment Estimation(Adam) [18].

$$\boldsymbol{v}_t = \beta_1 \boldsymbol{v}_{t-1} + (1 - \beta_1) \nabla f(\boldsymbol{\theta}_t) \tag{2.12}$$

$$\boldsymbol{m}_t = \beta_2 \boldsymbol{m}_{t-1} + (1 - \beta_2) (\nabla f(\boldsymbol{\theta}_t))^2 \tag{2.13}$$

It computes the first and second momentum $\boldsymbol{v}_t$,$\boldsymbol{m}_t$ term by computing a running average using the gradient or squared gradient respectively. β_1 and β_2 represent the decay factor of the running average.

$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - {\beta_1}^t} \tag{2.14}$$

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - {\beta_2}^t} \tag{2.15}$$

The completely new terms $\hat{\boldsymbol{v}}_t$ and $\hat{\boldsymbol{m}}_t$ reduce the bias of the running average at the beginning of the training because the at the beginning of the training the momenta are initialized with zero and are therefore biased towards zero. This adaption is therefore used to reduce the bias.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\boldsymbol{m}}_t + \varepsilon}} \hat{\boldsymbol{v}}_t \tag{2.16}$$

The algorithm essentially makes use of principles from the normal momentum term and from the second momentum to update the original parameters. In practice, it does quite

well in many cases [19] and will be the major optimization algorithm used in this work for training NNs.

2.1.3. Backpropagation

For every parameter update done by GD on the model parameters one needs the gradient of the empirical risk with respect to the model parameters. In Deep Learning models, can get very complex and deep, therefore the calculation of the gradient can get very complex. For those reasons and efficient algorithm for calculating the gradient is needed, otherwise using GD can become very costly for more than the shallowest models. The most used algorithm for calculating the Gradient is called Back Propagation(BP) which was first completely specified by [20]. The number of operations in BP is linear to the depth of the model O(N), where operations are basic operations of the neural network, for example matrix multiplication, addition of biases and activation functions. While a formal derivation is out of scope of this work, a short motivational example of how BP works will be given now. Suppose a small NN as in Figure 2.2 is given.

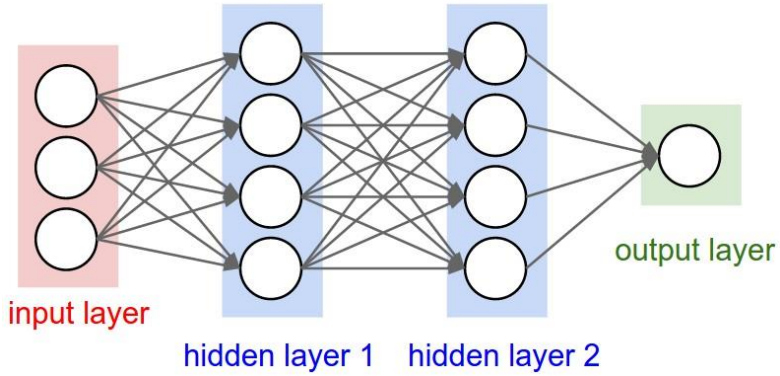

Figure 2.2 Multilayer Neural Network [21]

The equation to calculate the output of this NN is:

$$\boldsymbol{y}(\boldsymbol{x}) = f_o(\boldsymbol{W}_o f_h(\boldsymbol{W}_h f_i(\boldsymbol{W}_i \boldsymbol{x}))) \tag{2.17}$$

here $\boldsymbol{W}_o, \boldsymbol{W}_h, \boldsymbol{W}_o$ are the weight matrices of the input, hidden and output layer, and f_i, f_h, f_o are the respective activation functions. In this equation, the bias terms are embedded into the weight matrices and the size of every input vector into the matrix multiplication is increased by one with a constant element set to one. Now calculating the derivative of some loss $l(\boldsymbol{y}(\boldsymbol{x}), \boldsymbol{y}')$ with respect to $\boldsymbol{W}_o$and $\boldsymbol{O}_h = f_h(\boldsymbol{W}_h f_i(\boldsymbol{W}_i \boldsymbol{x}))$, by making use of the chain rule yields:

$$\frac{\partial l}{\partial \boldsymbol{W}_o} = \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{O}_o} \frac{\partial \boldsymbol{O}_o}{\partial \boldsymbol{p}_o} \frac{\partial \boldsymbol{p}_o}{\partial \boldsymbol{W}_o} \tag{2.18}$$

$$\frac{\partial l}{\partial \boldsymbol{O}_h} = \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{O}_o} \frac{\partial \boldsymbol{O}_o}{\partial \boldsymbol{p}_o} \frac{\partial \boldsymbol{p}_o}{\partial \boldsymbol{O}_h} \tag{2.19}$$

Here $\boldsymbol{p}_i = \boldsymbol{W}_i \boldsymbol{O}_{i-1}$ is the output of the product of the weight matrix in layer i and the output $\boldsymbol{O}_{i-1}$, after the activation function f_{i-1}, of layer $i-1$.

Now taking the derivative of the loss with respect to $\boldsymbol{W}_h$ and $\boldsymbol{O}_i$ yields:

$$\frac{\partial l}{\partial \boldsymbol{W}_h} = \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{O}_o} \frac{\partial \boldsymbol{O}_o}{\partial \boldsymbol{p}_o} \frac{\partial \boldsymbol{p}_o}{\partial \boldsymbol{O}_h} \frac{\partial \boldsymbol{O}_h}{\partial \boldsymbol{p}_h} \frac{\partial \boldsymbol{p}_h}{\partial \boldsymbol{W}_h} \tag{2.20}$$

$$\frac{\partial l}{\partial \boldsymbol{O}_i} = \frac{\partial l}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{O}_o} \frac{\partial \boldsymbol{O}_o}{\partial \boldsymbol{p}_o} \frac{\partial \boldsymbol{p}_o}{\partial \boldsymbol{O}_h} \frac{\partial \boldsymbol{O}_h}{\partial \boldsymbol{p}_h} \frac{\partial \boldsymbol{p}_h}{\partial \boldsymbol{O}_i} \tag{2.21}$$

But this is equal to, using equation (2.19):

$$\frac{\partial l}{\partial \boldsymbol{W}_h} = \frac{\partial l}{\partial \boldsymbol{O}_h} \frac{\partial \boldsymbol{O}_h}{\partial \boldsymbol{p}_h} \frac{\partial \boldsymbol{p}_h}{\partial \boldsymbol{W}_h} \tag{2.22}$$

$$\frac{\partial l}{\partial \boldsymbol{O}_i} = \frac{\partial l}{\partial \boldsymbol{O}_h} \frac{\partial \boldsymbol{O}_h}{\partial \boldsymbol{p}_h} \frac{\partial \boldsymbol{p}_h}{\partial \boldsymbol{O}_i} \tag{2.23}$$

This shows that the derivative of one layer can be computed by making use of the derivative of the loss with respect to the output of this layer which is computed before.

This principle generalizes to all layers a NN. This allows the calculation of the gradient in linear time in the number of layer by starting at the output of the network and then working your way back to the input of the neural network. Here also the name of the algorithm becomes clear, the inference step is also called forward pass because the calculation starts with an input and then proceeds to the output. The backward pass then calculates the gradient backwards by starting from the output and ending at the input. The whole algorithm is based on the repeated application of the chain rule.

Taking a closer look on the following calculation of the gradient with respect to the weight of an early layer shows a possible problem with the optimization of the model using the gradient. The gradient in deeper layers requires the multiplication of many derivatives. If many of the absolute gradients are either bigger than one or smaller than one the gradient might either explode or vanish. Those are the exploding and vanishing gradient problems [22], [23]. The exploding gradient problem can be solved relatively easy by introducing an upper limit on the gradient, so that it can't grow bigger than this upper bound. On the other hand, an introduction of a lower limit is not possible because it may be the case that the gradient is small or zero from the start. One way to reduce the problem of vanishing gradients is by choosing appropriate activation functions. The sigmoid activation function might be an ill choice because for small or big inputs into the function the gradient is near or completely zero which means that no gradient will be propagated further back. A good choice for an activation function is the ReLU function [24], [25]:

$$y=max(0,x) \tag{2.24}$$

The gradient is for values greater than zero constantly one but for values smaller than zero it is always zero. An additional benefit of the ReLU function is that it is very cheap to compute.

2.1.4. Convolutional Neural Networks

Convolutional Neural Networks(CNNs) are a special kind of neural feed-forward NN which were first proposed in [26] and later expanded up in more detail in [27] with the so called LeNet. They, where developed to solve vision related tasks such as classification of the content of an image and localizations of specific object for example faces. The input to an CNN is in general an image which may consist of one or multiple colour channels. The image is represented as a volume where the depth often depends on the number of channels, for example 3-dimensional, and the width and height depend on the size of the image. The CNN makes exhibits some very special properties present in images. One is translation invariance, which can be explained by the example of an object in an image. The object is the same class in general, class disregarding at which position it is located in the image. Another two invariances are the rotation and scaling invariance in images because the objects class stays the same disregarding the rotation and scaling of the object in the image. A NN that is used to classify objects in images and solves other tasks related to images should make use of such properties to increase its prediction accuracy. Normal NNs, as discussed before, don't make use of such properties because every pixel in the input volume is multiplied with a different weight. In order to extract the same information contained in the image at different locations, a normal NN has to learn the same weight multiple times which is quite inefficient. The solution to this

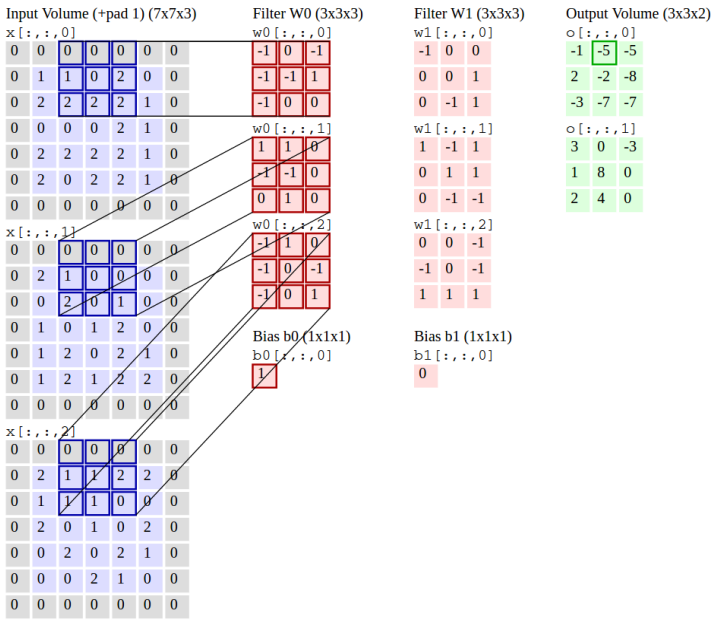

Figure 2.3 Convolution Example [28]

problem is a concept called weight sharing. Weight sharing allows the use of the same weights to calculate the input to different neurons and is used by CNNs. They make use of weight sharing in convolutional layers by multiplying the same weights with different patches of the image, therefore each convolutional layer extracts only local information from its input but the same information at many different positions. This operation is called convolution which is computed in the following way for one pixel in the following way:

$$\boldsymbol{S}_{i,j,k} = (\boldsymbol{x} * \boldsymbol{k})_{i,j,k} = \sum_{r}\sum_{s}\sum_{t} \boldsymbol{x}_{i+r,j+s,t}\boldsymbol{K}_{r,s,t,k} \tag{2.25}$$

$\boldsymbol{x}$ represents the input image volume to the convolution operation and $\boldsymbol{K}$ represents the so-called kernel or filter. In the case of a convolutional layer $\boldsymbol{K}$ consists of the parameters that are learned and shared by the output pixels because every output pixel in a feature map is computed using the same filter. The indices r and s are over the width and height of the filter and the index t goes over the depth of the input volume. Typical sizes for filters, in deep learning are 3×3, 5×5 but different sizes are also used [29]. Each image in the input or output volume is also called a feature map because it is an image that displays how strongly one feature, determined by the filter $\boldsymbol{K}$, is present in the input at all locations. It is useful to not just extract one feature but extract more than just one feature from the input volume. For this reason, each layer consists of many filters, hence the index k which goes over all output filters. The parameters of a convolutional are four dimensional objects while, the output of a convolutional layer is again three dimensional and called a feature volume. Figure 2.3 displays how the output of a convolution is computed for an example input volume and kernel. Each feature map also has a respective bias which is added after the convolution to every output in the feature map and each layer may have a non-linearity as in the case of normal NNs. The output of convolutional layer can be the input to another convolutional layer and so on. This results in a very important property of CNNs; While each layer makes only use of local information of its input it uses more global information of the networks input if the operation is performed after a few other operations were performed [5]. The reason for this is that each input pixel in, for example, the second layer already made use of local information of the input of the first layer. Let's assume the first layer has a filter size of thee and the second convolution also while, both have a stride of one. In this case, each output pixel of the second layer effectively depends on five input pixels of the first layer. This input block is also called the receptive field of the layer. Figure 2.4 visualizes this example. For those reasons, deeper levels in CNNs tend to extract more high-level features like whole eyes

or even faces, depending on the task and input data, while lower levels tend to extract more low-level features like edges and simple primitives. This also has some important implications regarding the computational cost of CNNs. The application of one convolution with a 7×7 filter size needs $49C^2$ operations, where C is the number of filter positions, in one dimension. Three convolutional layers, each with a filter of size 3×3, only needs $3(3^2C^2) = 27C^2$operations while still having the same receptive field, regarding the input to the first of the layer. Three consecutive convolutional layers with a filter size 3×3 and stride of one have a receptive field of 7×7. The benefit of three consecutive layers is that they are

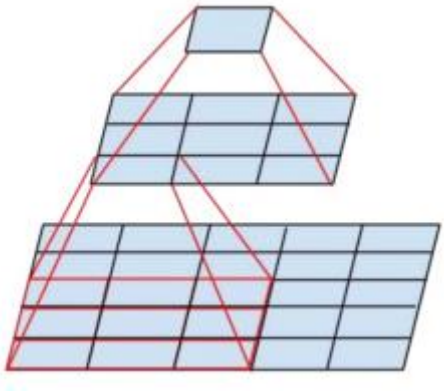

Figure 2.4 Receptive field of an CNN [30]

cheaper to compute and need less memory while, having the same receptive field. They also increase the non-linearity in the network, which might also be beneficial, because an activation function can be applied three times [31]. After all convolutions are performed the output of the last convolutional layer is often feed into one or multiple fully connected layer which are then used to, for example, extract probabilities of each class from the high-level features. The fully connected layers might be expressed by a matrix multiplication as was used in the normal NN but they might also be represented by convolutional layers, where the filter depth is the total number of pixels in the last feature volume and the number of filter maps is equal to the number of output neurons. In general, the size of each image/filter map is decreasing with increasing depth while the number of feature maps increases with the depth. The image size is reduced because in many cases the exact location of a feature is not as important and it also decreases the computational burden because convolutional layers with big images and many filters may be quite expensive. The reduction of the images size is often performed by so called pooling layers and there exist a few different ways in which they can archive this. In most cases pooling layers let a window stride over the input pixels then compute at each striding position an output pixel from the input pixels which fall in the window. If the stride is bigger than one a reduction of the image size takes place. The differences between each pooling type lies in how they compute the output pixel. Often used examples are Max pooling and Average pooling. Max pooling returns the maximal element in its receptive field at every

position. Average pooling takes the average of all elements in its receptive field. The benefit of such pooling operations is that they increase the spatial invariance of the network and the robustness [5]. Another approach to perform pooling is by using convolutional layers where each filter stride is bigger than one and by using the same number of filters as there are input feature maps. This means that is in principle possible to represent all major operations in an CNN as convolutions.

2.1.5. Recurrent Neural Networks

Feed-Forward NNs are quite powerful at solving task related to single images or other non-sequential data but they are not so good at working with sequential input or output.

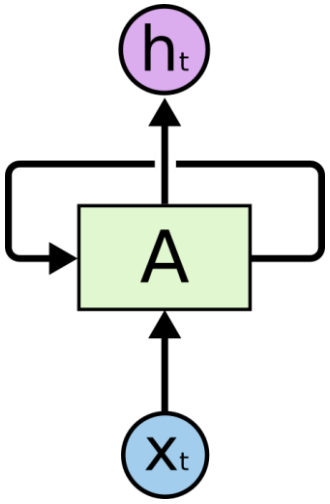

Figure 2.5 Recurrent Neural Network Layer [32]

For this reason, another class of NNs where developed, namely Recurrent Neural Networks (RNNs). RNNs are useful if either the input, output or both are sequential. Examples applications of RNNs are language translation, sentiment analysis of text and recognizing printed text from an image. They can solve these problems by introducing recurrent connections at some layers, an example can be seen in Figure 2.5 This means that the graph of the neural network contains circles. The recurrent connections introduce a state in the network which is the main difference between normal feed-forward NNs and RNNs [33]. The general equation for a recurrent layer is as follows [34]:

$$\boldsymbol{y}_t, \boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t) \tag{2.26}$$

Here $\boldsymbol{s}_t$ represents the state, $\boldsymbol{y}_t$ the output of the layer and $\boldsymbol{x}_t$ the input at step t. The most basic recurrent layer multiplies the state and the input with a parameter matrix, then adds the results and applies a non-linearity, in this example the tangents hyperbolics, to compute the output.

$$\boldsymbol{y}_t = \boldsymbol{s}_t = \tanh(\boldsymbol{W}\boldsymbol{x}_t + \boldsymbol{U}\boldsymbol{s}_{t-1}) \tag{2.27}$$

$\boldsymbol{W}$, $\boldsymbol{U}$ are the parameters with respect to which is the result optimized. In this case, the output is also used as state for the next time step. RNNs are theoretically Turing-complete but in practice it is very hard to learn even the simplest programs. To increase the capacity and depth of the network it is also possible to stack multiple layers of RNNs other each other.

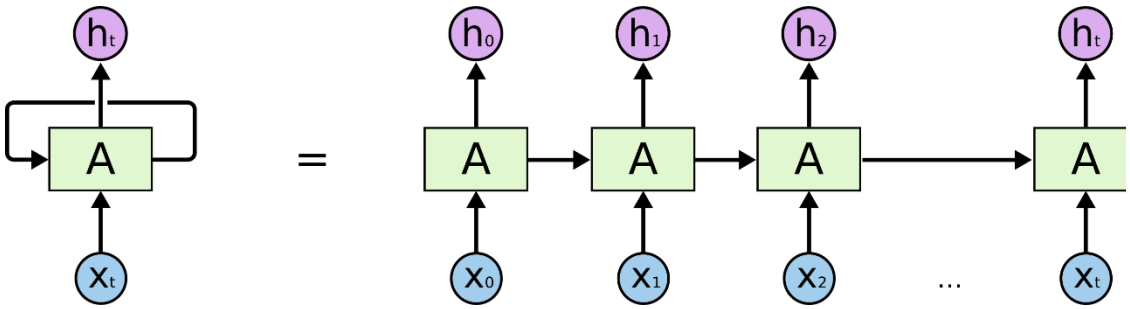

Figure 2.6 Unrolled Recurrent Neural Network [35]

2.1.6. Backpropagation trough Time

One question that arises when working with RNNs is how the gradient is computed because the neural network is now cyclical. This means that the gradient, with respect to the parameters of the NN, might depend on all time steps. For this reason, the training should be done on a finite number of time steps because all intermediate results must be stored and therefore the number of steps must be finite to be practical. If the number of time steps is then finite one may unroll the RNN over time [36] as is done in Figure 2.6 This new unrolled network has much similarity with a normal deep NN, with the exception that the depth depends on the number of time-steps and the parameters, in an unrolled neural network, are shared between many layers. The difference to normal NNs is that the unrolled NN has multiple outputs and inputs where, the number of outputs and inputs depends on the number of time steps. There are different ways in which the output of the RNN might influence the loss function. In the case of classification often only the output at the last time step is used to classify and all other outputs are discarded. In the case of sequence generation, the output at each time step may generate a new part of the sequence. The loss depends then on the total sequence of outputs. The algorithm used for computing the gradient of such networks is called Backpropagation trough Time (BPTT) [37]. The gradient is essentially computed by adding multiple gradients because the parameters are shared between many layers in the unrolled NN and the output at one time step might influence many different output time steps. If you unroll the NN in time to the right and the output goes up as displayed in Figure 2.6 Then it is quite easy to come up with how the algorithm works. In the Figure, the outgoing arrows represent operations and the nodes represent data. Then if two operations leave one node, some form of copy takes place, because two operations depend on the same input data. Therefore, the gradient of the loss with respect to this data node in the graph is:

$$\frac{\partial l}{\partial \boldsymbol{d}} = \frac{\partial l}{\partial \boldsymbol{o}_1}\frac{\partial \boldsymbol{o}_1}{\partial \boldsymbol{d}} + \frac{\partial l}{\partial \boldsymbol{o}_2}\frac{\partial \boldsymbol{o}_2}{\partial \boldsymbol{d}} \tag{2.28}$$

Here $\frac{\partial l}{\partial \boldsymbol{o}_1}$, $\frac{\partial l}{\partial \boldsymbol{o}_2}$ represents the gradient of the loss with respect to the output of both operations $\boldsymbol{o}_1, \boldsymbol{o}_2$, that use this data and the terms $\frac{\partial \boldsymbol{o}_1}{\partial \boldsymbol{d}}$, $\frac{\partial \boldsymbol{o}_2}{\partial \boldsymbol{d}}$ are the gradient of the two operations with respect to the data. To compute all gradients one needs to compute the gradient the same way as done in BP but starting from the top right in the unrolled NN and then first calculate all gradients from up to down, as in the normal NN case, and then from left to right. At the nodes where the data is split the gradients of the outgoing operations are added together to compute the total gradient with respect to this data. This is especially true for gradients with respect to the parameters, of the NN, in the recurrent layers because those are essentially copied number of times steps times. This way one obtains an algorithm that can compute the gradient linear in the depth of the original network and linear in the number of time steps. The total time complexity is $O(t_{max}d_{max})$, where t_{max} is the number of time-steps and d_{max} the maximal depth of the original unrolled NN.

One problem of BP and BPTT is that all intermediate results of the forward pass must be stored to be able to compute the gradients. This has a big influence on the performance of the algorithm because memory might be the limiting factor if the number of time-steps is too big or the depth is too high. One way to reduce the memory requirements is by storing not all intermediate results but only a few of them and then use them to re-compute the missing results, this can be done in an optimized fashion [38].

In the case of RNNs it is very important to lock at the case of long-term dependencies. These are situations where an early input potentially has a big impact on output of a later time-step. The gradient of the last output with respect to the first input is calculated by multiplying many gradients. This can be seen by taking the path from the last output to the first input, in Figure 2.6 and multiplying all gradients that belong to the operations taken on this path. This results in following equation:

$$\frac{\partial \mathrm{l}}{\partial \boldsymbol{x}_0} = \frac{\partial \boldsymbol{s}_o}{\partial \boldsymbol{x}_0}\prod_{t=1}^{t_{max}}\frac{\partial \boldsymbol{s}_t}{\partial \boldsymbol{s}_{t-1}}\frac{\partial l}{\partial \boldsymbol{s}_{t_{max}}} \tag{2.29}$$

If many of the, absolute value, of those gradients are bigger one the gradient increases exponentially. This is the same problem as before called the exploding gradients and can be solved by clipping the maximal gradient. The bigger issue arises if many of the absolute values, of the gradients, are smaller than one. In this case, the gradients decay exponentially. This is again the problem of vanishing gradients as was stated before. This

problem has a major impact on the performance of RNNs because long-term dependencies might not be learned because the gradients with respect to the first time-steps might be very small compared to the gradients of later time-steps [23]. Hence, the update on the parameters over those long-term dependencies is neglectable compared to the short-term dependencies.

Two ways on how to reduce the influence of this problem will be explained in the following sections.

2.1.7. Long Short-Term Memory and Gated Rectifier Unit

The Long Short-Term Memory (LSTM) is a special form of a recurrent cell and was first developed in [39] and was introduced to reduce the problem with long-term dependencies in the RNN setting. To reduce the vanishing gradient problem and the exploding gradient problem following equation needs to be fulfilled [40]:

$$\frac{|\partial \mathbf{s}_{t+k}|}{|\partial \mathbf{s}_t|} \leq \|\gamma \boldsymbol{W}\|^k = 1 \tag{2.30}$$

Here $\gamma = \sup(\|f'\|)$ is the supremum of the gradient of the activation connecting one state to the state of the next step and $\boldsymbol{W}$ is the weight that connects them. If the product is greater than one the gradient explodes while if the product is smaller than the 1 the gradient will vanish. The LSTM solves this problem by a gating mechanism and because the activation function is essentially the identity. The equations for the LSTM are the following:

$$\boldsymbol{i} = \sigma(\boldsymbol{x}_t \boldsymbol{U}_i + \boldsymbol{s}_{t-1} \boldsymbol{W}_i) \tag{2.31}$$

$$\boldsymbol{f} = \sigma(\boldsymbol{x}_t \boldsymbol{U}_f + \boldsymbol{s}_{t-1} \boldsymbol{W}_f) \tag{2.32}$$

$$\boldsymbol{o} = \sigma(\boldsymbol{x}_t \boldsymbol{U}_o + \boldsymbol{s}_{t-1} \boldsymbol{W}_o) \tag{2.33}$$

$$\boldsymbol{g} = \tanh(\boldsymbol{x}_t \boldsymbol{U}_g + \boldsymbol{s}_{t-1} \boldsymbol{W}_g) \tag{2.34}$$

$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} \odot \boldsymbol{f} + \boldsymbol{g} \odot \boldsymbol{i} \tag{2.35}$$

$$\boldsymbol{s}_t = \tanh(\boldsymbol{c}_t) \odot \boldsymbol{o} \tag{2.36}$$

$\boldsymbol{f}$ is called the forget gate, $\boldsymbol{i}$ is called the input gate, $\boldsymbol{o}$ the output gate, $\boldsymbol{s}_t$ is the output of the cell and $\boldsymbol{c}_t$ is the memory of the cell at time-step t. This is the extended version of the algorithm and not the first proposed version. There exist many other versions but we will focus on this one as it is often used. Here one can see that the activation function that connects one memory to the next is the identity meaning it is always one and the effective

weight that is multiplied to the state and $\boldsymbol{c}_t$ is the forget gate output. This output is in the range 0 -1 and hence the equation can be fulfilled allowing the gradient to flow undiminished preventing the vanishing and exploding gradient problem. Further analysis of the equations given a deeper insight of the function of the different gates and why this architecture might be beneficial. Looking at equation (2.35) one can see that the input gates output is multiplied element-wise with some transformation of the input. This means that the input gate chooses which part of the input is added to the next state and therefore saved in the state, archiving something like a selective read of the input. The other part of the equation is the element-wise product between the forget gate vector and the last state which represents some kind of selective memory, because only parts of the last state remain while some other parts are overwritten. The last gate, the output gate is essentially a selective write, since it decides, which parts of the transformed state are used as output in equation (2.36). They also have another interesting effect compared to the normal RNN cell. While the classical RNN transforms the information, which is saved in the state, every time-step, the LSTM is principally able to keep the information saved in the state untransformed for multiple time-step and it is able to only change parts of it. One problem that can occur with LSTMs is that the state can blow up over multiple time-steps because the left part of the sum in equation (2.35) can potentially be bigger than one, while the right part can be maximally one possible resulting in an increase of c at every time-step. An alternative RNN cell, which is able to solve this problem, is the Gate Rectifier Unit (GRU) [41]. The GRU cell calculates the state and the output in the following way:

$$\boldsymbol{z} = \sigma(\boldsymbol{x}_t \boldsymbol{U}_z + \boldsymbol{s}_{t-1} \boldsymbol{W}_z) \tag{2.37}$$

$$\boldsymbol{r} = \sigma(\boldsymbol{x}_t \boldsymbol{U}_r + \boldsymbol{s}_{t-1} \boldsymbol{W}_r) \tag{2.38}$$

$$\boldsymbol{h} = \tanh(\boldsymbol{x}_t \boldsymbol{U}_h + (\boldsymbol{s}_{t-1} \odot \boldsymbol{r}) \boldsymbol{W}_h) \tag{2.39}$$

$$\boldsymbol{s}_t = (1 - \boldsymbol{z}) \odot \boldsymbol{h} + \boldsymbol{z} \odot \boldsymbol{s}_{t-1} \tag{2.40}$$

here $\boldsymbol{z}$ is the update gate vector, $\boldsymbol{r}$ the reset gate vector, $\boldsymbol{s}_t$ the state vector at time step t and $\boldsymbol{h}$ is the output at time step t. The GRU cell is also a gated cell while the computation of the output is simpler compared to the LSTM. It needs less matrix multiplications and needs to store less intermediate results while, the GRU cell has the same benefits regarding the vanishing gradient problem. One major difference is the fact that the new state is computed as a weighed sum of the last state and some transformation of the input. This way the state does not increase because the right part of the sum in equation is always

smaller or equal to one while the left part is, if it is initialized with values smaller or equal to one, also never bigger than one. This way the weighted average stays smaller or equal than one hence never exploding. The z vector decides how much of which part of the state remains the same and how much it is changed. The r vector decides how much of the old state is added to the input. This cell type potentially also allows the information saved at one time-step to stay unchanged. The GRU cells also have less parameters which can potentially reduce the tendency of the model to overfit. A direct comparison of all three cell types, basic tanh RNN cell, LSTM and GRU, show that the GRU cell is in many cases superior over the LSTM and basic tanh RNN cell [42].

2.1.8. Connectionist Temporal Classification Loss

One problem that can arise when working with RNNs is that there might be no clear alignment between the input sequence and the expected output sequence. An example is voice recognition in which the input signal can be split up into windows which are then feed as input to the RNN. In this case, different uttered words or word parts are of different length therefore making it not clear if a full word/ word part is contained in the window or only a part of it. For example, the signal of a word might be split into two different windows and it is not clear at which window should the RNN output the word and what should it output at the other window position. The solution to this problem was introduced in [43] and is called Connectionist Temporal Classification (CTC) Loss. It allows the system to handle such situations by introducing a special transformation $\boldsymbol{B}(\boldsymbol{\pi})$, a specific loss and a way how to handle the output of the RNN in a specific way. The first determines how the output of the RNN is handled by transforming the original output string $\boldsymbol{\pi}$ of the RNN to a new one. This is done by merging multiple consecutive outputs of the same class to only one output, effectively reducing the size of the output word. The following example shows how this is done for the word “You”.

$$\text{YYouuu} \quad \rightarrow \quad \text{You}$$

One problem with this approach is that when the same class must be returned multiple consecutive times as is the case for the word Hello:

$$\text{Heeellllloo} \quad \rightarrow \quad \text{Helo}$$

This is solved by the introduction of an additional class that the system can output, called the *blank* class. When this class is returned, the transformation just removes it from the sequence but acknowledges that the following element in the sequence is a new return of the class therefore allowing the transformed sequence to contain the same class multiple

times, in sequence, by inserting one or multiple *blank* between them. The following example shows how Hello can be created this way, the blank is represented by <*blank*>:

Heeel<*blank*><*blank*>llloo → Hello

The *blank* class can also be used when the NN does not want to output a specific class, by outputting a *blank* which is just erased in the transformation. The sequence for generating a specific sequence, after the transformation is not unique:

Yoouu → You <*blank*>YYouu → You Yo<*blank*>u<*blank*> → You

The goal of the Training would therefore be to minimize the logarithmic probability,

$$\log(p(\boldsymbol{l}|\boldsymbol{x})) \tag{2.41}$$

of the correct label sequence $\boldsymbol{l}$ after the transformation on the output for each trainings example $\boldsymbol{x}$. Many different RNN output sequences are transformed to the same total output therefore the probability of the correct output can be computed in the following way:

$$p(\boldsymbol{l}|\boldsymbol{x}) = \sum_{\boldsymbol{\pi}\epsilon\boldsymbol{B}(\boldsymbol{l})^{-1}} p(\boldsymbol{\pi}|\boldsymbol{x}) \tag{2.42}$$

Here $\boldsymbol{\pi}$ denotes all possible paths, which can be created by the output of the RNN, that can be transformed to the correct string. The computation of this must be performed over all paths $\boldsymbol{\pi}$ that create the label sequence which can be quite expensive. There might be many paths $\boldsymbol{\pi}$ which get transformed to the label sequence and it must be executed for every example $\boldsymbol{x}$ of the training set. To make the computation efficient a specific forward backward algorithm was introduced which is inspired by one forward backward algorithm that is used for Hidden Markov Models [44]. The Idea behind the algorithm is to split the computation of the probability into two different parts, the forward pass, and the backward pass, which can be computed in a recursive way. Between each element, at the beginning and at the end, is a blank element added allowing either to switch from one label to the next one in the sequence or to the blank label followed by a switch to the next real label sequence. The forward pass then computes the probabilities $\alpha_t(s)$ of the sub sequence $\boldsymbol{l}`_s$ until label s at time step t, starting from the beginning of the sequence.

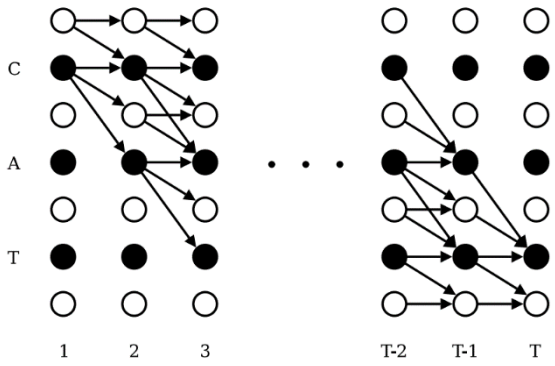

Figure 2.7 CTC Forward Pass [43]

A benefit of this approach is that the probability each time step and sequence length only depends on the probability of the last time step with the same sequence and sequence reduced by one or two elements. The reason is that a time steps can only reach elements in the sequence that are maximal two elements in the sequence away. At one time step, it can only output the same label it had before, it can output a blank or the next correct label. All other possibility would result in a wrong path. Figure 2.7 displays this behavior. The recursive algorithm computes the probabilities in the following way:

$$\alpha_t(s) = \begin{cases} \overline{\alpha_t}(s) y^t_{\boldsymbol{l}`_s} \text{ if } \boldsymbol{l}`_s = blank \text{ or } \boldsymbol{l}`_{s-2} = \boldsymbol{l}`_s \\ \left(\overline{\alpha_t}(s) + \alpha_{t-1}(s-2)\right) y^t_{\boldsymbol{l}`_s} \text{otherwise} \end{cases} \tag{2.43}$$

$$\overline{\alpha_t}(s) \stackrel{\text{def}}{=} \alpha_{t-1}(s) + \alpha_{t-1}(s-1) \tag{2.44}$$

With the following initial conditions:

$$\alpha_1(0) = y^1_{blank} \tag{2.45}$$

$$\alpha_1(1) = y^1_{\boldsymbol{l}`_1} \tag{2.46}$$

$$\alpha_1(s) = 0, \forall s > 2 \tag{2.47}$$

Here $y^t_{\boldsymbol{l}`_s}$ for the probability that is returned by the RNN for label s at time step t. The computation of the backward pass works the same way but starting at the end and works then in the opposite direction. The algorithm also incorporates some rescaling because otherwise this algorithm would often tend to underflow [44]. Multiplying both results at some arbitrary time step and label results in the probability of all paths of $\boldsymbol{l}$ going through s at time step t. The gradient with respect to the RNN output can then be calculated using the forward and backward variables [43].

$$-\frac{\partial \ln(p(\boldsymbol{z}|\boldsymbol{x}))}{\partial \boldsymbol{a}^t_k} = \boldsymbol{y}^t_k - \frac{1}{p(\boldsymbol{z}|\boldsymbol{x})} \sum_{s \epsilon lab(\boldsymbol{z},k)} \hat{\alpha}_t(s)\hat{\beta}_t(s) \tag{2.48}$$

$\boldsymbol{z}$ represents the target label sequence, $\boldsymbol{a}^t_k$ represents the activation of neuron k, at time step t, of the layer leading into the Softmax. $\hat{\alpha}_t(s)$ and $\hat{\beta}_t(s)$ are the normalize forward and backward variable and $lab(\boldsymbol{z}, k) = \{s : \boldsymbol{l}`_s = k\}$. One important aspect that must be considered is how the output of the system at inference time is calculated because there are two ways to do this. One is to sample the sequence by taking the maximum probability after a Softmax at each time step. The alternative would be to compute the labeling sequence with the highest probability after the transformation which can be approximated using beam search. The second approach tends to achieve better results and the CTC loss

has proven to be quite successful in practice also for handwritten character recognition [45].

2.1.9. Encoder Decoder

To use RNNs one needs to consider things like how is the input to the RNN handled and how is the state initialized. One example task that one might try to solve is the task of translating a sentence in one language to another language. The question is how is the input to the RNN passed and which output should be used. It would be possible to use one word as input at every time step and then output a word in the translated language. The problem with this is that to translate a sentence accurately, information from the complete sentence might be necessary and not only one word or all words until the current time step. A solution to this was presented in [46]. Instead of taking the output directly after one word was inserted, it is delayed until all words where passed as input to the RNN. Only then the output of the RNN is used and the first word of the translation is expected. This way when the RNN starts outputting the translated sentence it has the information from the complete sentence available encoded in its state. This means that the first steps of the RNN serve to encode the input sentence, transforming it into a fixed length vector in its state, followed by a decoding of the encoded sentence in the sentence of the translated language (Figure 2.8). This procedure could also be understood by having two different RNNs. The first RNN is then used to create an encoding of the input sentence which is then used to initialize the state of the second RNN. The second RNN uses this encoding in its state to decode it into the same sentence but in a different language. This approach of encoding and decoding, was translated to the situation of image captioning [47]. Image Captioning aims at generating a short description of the image content. The input to the NN is therefore an Image and the output is a sentence. Instead of creating an encoding of an input sentence using an RNN one creates an encoding of an input image using a CNN (Figure 2.9). The

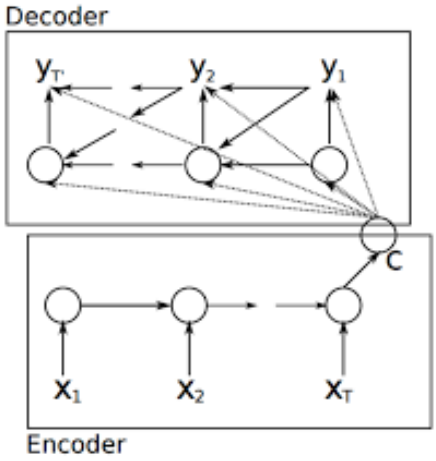

Figure 2.8 Encoder Decoder Translation [46]

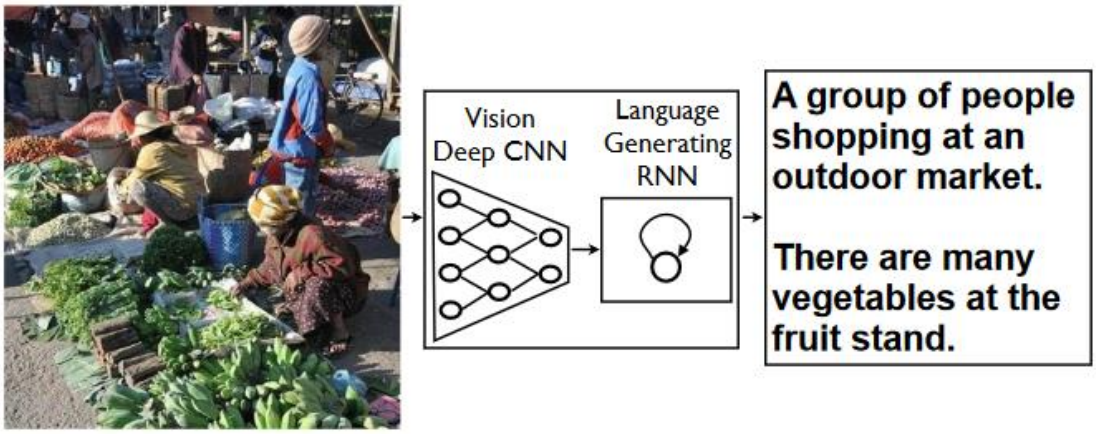

Figure 2.9 Encoder Decoder Image [48]

encoding can then be used to initialize an RNN. This way the strengths of both, CNNs for images and RNNs for sequences, can be exploited.

2.1.10. Attention Mechanisms

In the Encoder Decoder architecture, the RNN receives some transformed version of the input data only at the first step as initialization of the RNN and then never again. This makes it necessary that the information is stored long enough and enough important information is stored. This can be quite problematic for long sequences as the information must be stored for quite some time and the amount of necessary information might be too big. It also restricts the possibility's how the performance of the system can be increased further since the major bottleneck is the encoding. A solution to this was introduced in [50] by adding a so called soft attention mechanism. An encoding of the input is generated in some way but, instead of initializing the state of an RNN with it, it is used to create a context vector $\boldsymbol{c}_i$ at each time step i which is then feed as input to the RNN. An encoding for translating language can be generated by applying a bidirectional RNN on the input sentence and taking the output of each time step as encoding (Figure 2.10), which is also called a sequence of annotations $(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{T_x})$. The sequence of annotations is used to generate an attention vector:

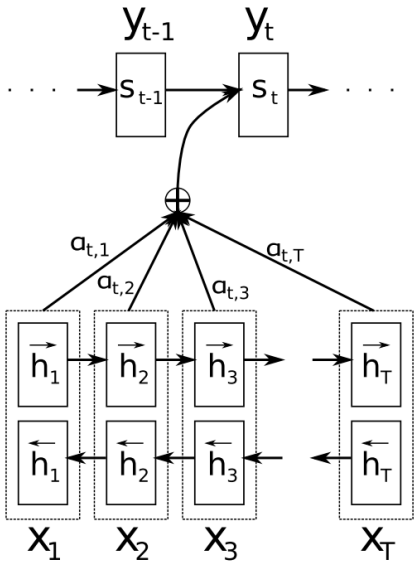

Figure 2.10 Attention Example [49]

$$\boldsymbol{\alpha}_{i,j} = Attention(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j) \tag{2.49}$$

With the state $\boldsymbol{s}_{i-1}$ of the RNN at step $i-1$ and annotation vector $\boldsymbol{h}_j$. This is also called "content-based" attention because the attention is computed using the encoded content of the input. There also exist also a so called "location-based" attention mechanism [51] which computes the attention based on the last attention vector:

$$\boldsymbol{\alpha}_{i,j} = Attention(\boldsymbol{s}_{i-1}, \boldsymbol{\alpha}_{i-1,j}) \tag{2.50}$$

Both approaches can also be combined to acquire a hybrid attention mechanism [52] which makes use of the last locations and the input:

$$\boldsymbol{\alpha}_{i,j} = Attention(\boldsymbol{s}_{i-1}, \boldsymbol{\alpha}_{i-1,j}, \boldsymbol{h}_j) \tag{2.51}$$

The attention vector $\boldsymbol{\alpha}_{i,j}$ is used to calculate the context vector $\boldsymbol{c}_i$ as a weighted sum:

$$\boldsymbol{c}_i = \sum_{j=1}^{T_x} \boldsymbol{\alpha}_{i,j} \boldsymbol{h}_j \tag{2.52}$$

This vector is used as input to the RNN which generates the output sequence. The calculation of the attention vector is performed in two steps. The first part $\boldsymbol{e}_{i,j}$, the alignment score, is calculated using a simple feed forward Neural Network, in the case of "content-based" attention, for example:

$$\boldsymbol{e}_{i,j} = \boldsymbol{v}_a \tanh(\boldsymbol{W}_a \boldsymbol{s}_{i-1} + \boldsymbol{U}_a \boldsymbol{h}_j) \tag{2.53}$$

$\boldsymbol{v}_a$, $\boldsymbol{W}_a$, $\boldsymbol{U}_a$ are parameters that are trained together with the rest of the NN. The alignment score is then used to calculate the alignment vector by normalizing the exponentials of it, over annotation vector length $T_x$which is equal to the length of the input vector.

$$\boldsymbol{\alpha}_{i,j} = \frac{\exp(\boldsymbol{e}_{i,j})}{\sum_{k=1}^{T_x} \exp(\boldsymbol{e}_{i,k})} \tag{2.54}$$

This encourages the NN to focus on some of part of the input annotation vector. The attention mechanism has proven to be quite beneficial for machine translation. It learned to concentrate on important parts of the input and disregard information's that where not so important on the current time step. This makes the results also better interpretable because one can observe which parts of the input are deemed important by the NN. The attention mechanism was successfully applied to images for caption generation [52]. In this situation, the annotation vectors are the different pixels in the input image and each annotation vector has the number of feature maps, elements. The attention can be performed over each layer output of the CNN. Since the attention vector is now a three-

dimensional object, it will be called an annotation map, conforming to the lingo of CNNs. The attention allows the user to interpret where the neural network is looking at, to generate the current output word, therefore making it more possible what to understand what the NN thinks of as being important. Attention was also successfully applied to handwritten character recognition in [53]. This work used a Multi-Dimensional Long Short-Term Memory RNN to compute the attention score and not a normal feed-forward NN as was done before.

2.2. Programming Concepts and OpenCL

In this section, a short introduction to OpenCL and how GPUs are programmed will be given, followed by an introduction to the algorithms that are used to implement the operations on the GPU. It starts with an introduction to OpenCL followed by an introduction to concepts of GPU programming. Afterwards a short introduction to tile based algorithms for GPU programming will be given. A special collection, called a Tuple requires special attention and will therefore be explained afterwards. The last part than introduces the Image to Columns algorithms for Convolutions.

2.2.1. OpenCL

OpenCL is a standard for the parallel programming of heterogeneous hardware which was agreed upon by the Khronos group [54] [55] which is non-profit consortium consisting of many members. The current OpenCL version is 2.2. It was developed to create a standard interface to programming different parallel devices. It is supported by many different hardware manufactures for example Intel [56], NVIDIA [57], AMD [58] and Xilinx [59], therefore it supports a wide range of different devices. Supported devices are for example GPUs, CPUs, integrated GPUs and FPGAs. While many different manufactures support it not all of them support the newest version. The newest NVIDIA GPUs for example supports only OpenCL1.2 while Intel's 7th generation processors support OpenCL 2.1. [56] This restrict the portability and development of programs making heavy use of OpenCL because one must decide beforehand what kind of hardware should be supported and therefore restrict the features that are available to the developer. OpenCL consists of two parts [60]. The first the API is used to control the hardware that conducts parallel processing this part runs on the host system. The second part is a C99 or for newer versions C++ style programming language for creating programmes that run on the device used for parallel programming. Every implementation comes with its own compiler for compiling the programs. The programs that run on the device are also called kernels and need to be defined by using the kernel keyword after the return value

declaration. Even though the code that is created for the device is in principle portable, assuming it adheres to the supported version, it is often necessary to adapt the code to new devices because every type of hardware has different requirements on how the code needs to be optimized. One example is the difference between GPUs and CPUs. CPUs haven't so much calculating units compared to GPUs and therefore can't parallelized the code as much but are very effective at program branching while GPUs need many threads to be active at the same time to be effective. Even if a kernel program runs on multiple devices it can be so slow that the benefit of using parallelized code vanishes without optimized for the hardware.

2.2.2. GPU programming concepts with OpenCL

Graphics Processing Units in short GPUs are optimized for high parallel throughput while having relatively low clock speeds compared to CPUs. They are especially optimized for many parallel floating-point operations because the original purpose of GPUs was the processing of graphics which makes major use of floating point operations. OpenCL

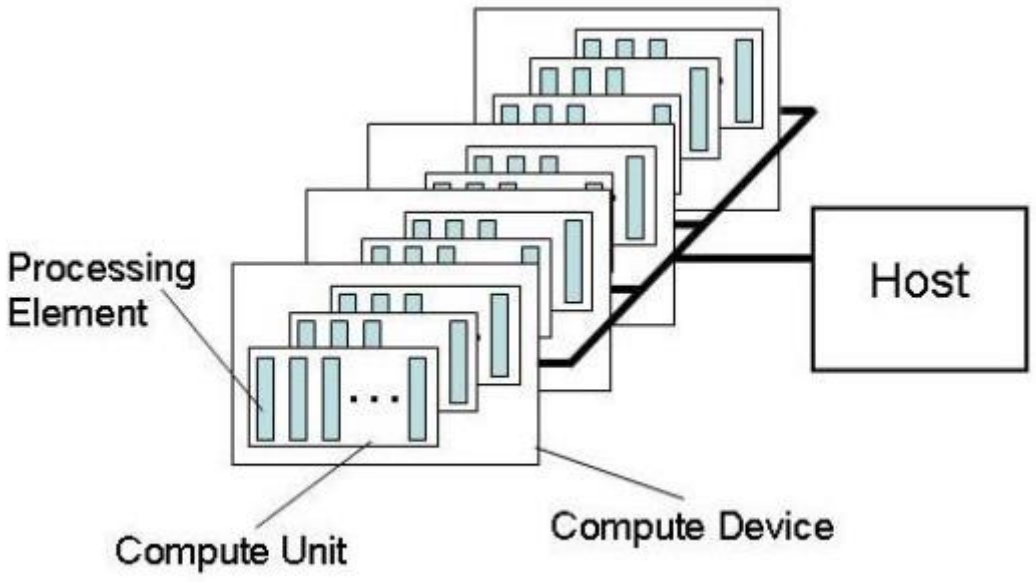

Figure 2.11 OpenCL Compute Model [55]

thread execution model splits the parallel processing capabilities into a hierarchy [55]. The host is used to control one or multiple compute devices. Those are in general the available hardware devices like GPUs, integrated CPUs and so on. This hardware devices contain multiple compute units which then contain multiple processing elements this structure can be seen in Figure 2.11. What those mean in the sense of the physical available hardware is different from hardware to hardware. The work that needs to be done is split up again into multiple work-groups which consist of multiple work-items.

The functions called kernels run then on the single work-items. The total work is represented by an n dimensional grid which is then again split up into n dimensional work-groups. This means that the size of the total work in every dimension must be a multiple of the size of the corresponding dimension in the work-group [60]. This order of how threads are executed have implications on how the threads can be synchronised. While the threads cannot be synchronised on a intra work-group level, it is possible to perform synchronisation within a work-group. The OpenCL memory layout specifies different regions of memory which are the global memory, the constant memory, the local memory and the private memory, all of them can be seen in Figure 2.12.

The global memory is accessible by all work regions and allows read and write operations to it, OpenCL also allows the accesses to the global memory to be cached if the device has appropriate capabilities. The constant memory is a part of the global memory from which work-items are only allowed to read and is initialized by the host. The local memory is shared by a work-group and allows the creation of variables that are used by all work-items in a work-group and allows synchronisation between them. Local memory

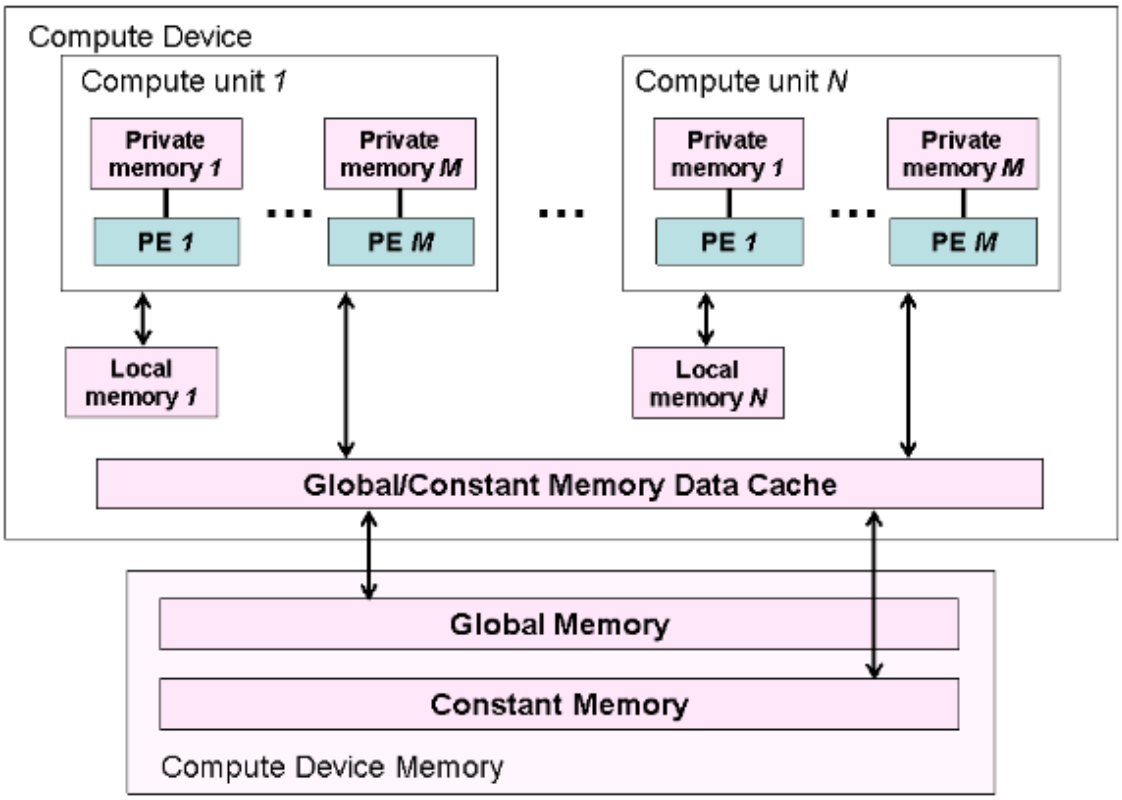

Figure 2.12 OpenCL Memory Model [55]

can only be written to by the kernel and not the host program therefore making explicit memory writes into in necessary when it should be used. Private memory is a memory region, that is private to single work-items and only the respective work-item is able to

access it. In order to know, how the kernel objects are best optimized, one has to know how the OpenCL concepts will be mapped to the device. For this reason, a short explanation of how the concepts map to NVIDIA GPUs will be made. The mapping from OpenCL to AMD GPUs is quite similar and this work was primarily performed using NVIDIA GPUs therefore the concentration on NVIDIA GPUs. NVIDIA GPUs contain multiple streaming multiprocessors which contain multiple so called CUDA cores, the actual number of the streaming multiprocessors and the CUDA cores depends on the actual device. The CUDA cores execute the work items and work-groups are executed on the streaming multiprocessors. Work-groups are divided into so called warps which are 32 threads big [61]. All threads essentially perform the same instruction at the same time this also is the reason why it is necessary to minimize the branching in the program. If the program contains much branching many control branches must be fully computed even though they are not needed therefore needed much more operations for the kernel execution. One streaming multiprocessor can have more than one warp allowing latency hiding because while one warp is blocked another one can run on the GPU. Meaning that work-groups should be big enough to create more warps and therefore allow better latency hiding. The size of work-group should be a multiple of the warp size otherwise some CUDA cores might be unused. The streaming processors contain shared memory, a register which allow a mapping from local memory to shared memory and from private memory to registers. They also contain also global memory and an L2 allowing the caching of accesses to the global memory. From this architecture, one can see that while access to local memory is quite fast, the access to global is relatively slow and should therefore be minimized. If the global memory is accessed in a coalesced fashion by a half warp the read or write take only one or two transactions in the case of 128-bit words. The requirements for coalesced global memory accesses depend on the compute capabilities of the device. For all following explanations, we assume that the used device has compute capability 1.2 or higher because the current generation of NVIDIA GPUs supports compute capability 6.1 and only first generation CUDA has compute capability 1.x. To understand coalesced access, it is assumed that the memory is comprised of aligned 64 or 128-byte segments. To be coalesced the access of one half warp must fully pass into one 128-byte segment for 32, 64 and 128-bit data, for smaller data reads the segment must be smaller respectively but behaves otherwise the same. Important is the fact that the concrete access pattern, if it lies in the 128-byte segment is not important [61]. If the memory region that is accessed by the threads of a half warp is sequential but falls into 128-byte segments of memory two access transactions need to be performed.

Another important aspect of GPU programming that was neglected until now is the transfer of data from the host CPU to the GPU and backward. This aspect is important because the transfer of data from the CPU to the GPU and backward is very slow and therefore should be minimized as much as possible.

```
template <class... Ts> struct Tuple {};

template<class T, class... Ts>
struct Tuple<T, Ts...> : Tuple<Ts...>
{
	Tuple(T t, Ts... ts) : Tuple<Ts...>(ts...), value(t) {}
	Tuple() : Tuple<Ts...>(), value(){}
	T value;
};
```

Code 2.1 Tuple Class

2.2.3. Tuples

A Tuple is a list of elements where the elements don't need to be of the same type but can

```
template<size_t, class> struct ElemHolder;

template <class T, class... Ts>
struct ElemHolder<0, Tuple<T, Ts...>>
{
	typedef T type;
};

template <size_t k, class T, class... Ts>
struct ElemHolder<k, Tuple<T, Ts...>>
{
	typedef typename ElemHolder<k - 1, Tuple<Ts...>>::type type;
};
```

Code 2.2 ElemHolder Class

essentially be anything Tuple [62] [63]. The possible types that can be stored, range from basic data types like integer to more complex things like class objects. The support of such objects was added in C++11 [63] by adding so called Variadic Templates. Variadic Templates allow functions and classes to have a variable number of templates and therefore, also a variable number of parameters. An example implementation of a tuple can be seen in Code 2.1. The Tuple requires the creation of an empty class followed by a specialized class with an arbitrary Variadic Template argument. A specialized class needs

RUB

then to be used to slice one template argument off the Variadic template. The sliced of template argument is used as member variable of the class. This serves to store the first value of the tuple. The specialized class contains a constructor which get the number of Variable templates arguments, parameters passed to it. The first parameter is also sliced off using the sliced of template argument. Variadic templates are stored in a hierarchy of derived classes, where, in this case, a base class contains one less template argument.

```
template<size_t k, class... Ts>
typename std::enable_if<k == 0, typename ElemHolder<0, Tuple<Ts...>>::type&
>::type
inline get(Tuple<Ts...>& t)
{
	return t.value;
}

template<size_t k, class T, class... Ts>
typename std::enable_if<k != 0, typename ElemHolder<k, Tuple<T, Ts...>>::type&
>::type
inline get(Tuple<T, Ts...>& t)
{
	Tuple<Ts...>& base = t;
	return get<k - 1>(base);
}
```

Code 2.3 Get Function

```
template<size_t current, size_t to, class... Ts>
struct Loop
{
public:
	inline static void apply(Tuple<Ts...> tuple)
	{
		FunktionObject<ElemHolder<current,
Tuple<Ts...>>::type>::func(get<current>(tuple));
		Loop<current + 1, to, Ts...>::apply(tuple);
	}
};

// terminal case
template<size_t current, class... Ts>
struct Loop<current, current, Ts...> {
public:
	inline static void apply(Tuple<Ts...> tuple)
	{
		FunktionObject<ElemHolder<current,
Tuple<Ts...>>::type>::func(get<current>(tuple));
	}
};
```

Code 2.4 Loop Class

Therefore, the constructor calls its base constructor with one less template argument in its base constructor and stores the sliced of parameter in the sliced of value member. This is done, recursively, until the Variadic template contains no more arguments. In this case the default Tuple class is called, which has a default constructor that does nothing. This way each derived class adds a new member variable to store an additional parameter. This means that Tuples can't really grow or shrink at runtime because the size must be defined at compile time. There remain two things that are needed to be able to make practical use of Tuples. The first is the question how elements are accessed. The second is how should one loop over the number of templates arguments, to perform some operations on them. It is not possible to write a dynamic for or while loop that iterate over all elements because the access to the elements must be done using template objects and arguments. To access a specific element of the Tuple again two things are necessary. The first is a way to iterate over the elements and retrieve the correct one. The second is to be able to get the return type, because a function that returns an element of the Tuple must be defined with the correct return type. The return type is determined by a struct template called ElemHolder that defines a typename type member. It has two template arguments passed to it, the index of the element for which the type should be returned and the variadic template of the tuple. It can be seen in Code 2.2.

It again has specialization that splits of the first element of the variadic tuple. If the index is zero, the type is set to the current split of template argument. Otherwise the type of a ElemHolder base class is queried, using the remaining variadic template and the index reduce by one. This is done recursively until the index is zero and the currently split off template is set to the type using typedef. The type member can then be used to query the type of an arbitrary element of the tuple automatically. The function get, that returns an element of the tuple can be seen in Code 2.3.

It has again the index and the variadic template passed to it as templates and the Tuple as parameter. It again calls itself recursively, slicing of one template each time and reducing the index by one. The tuple is reduced by one element each time. Therefore, each time a derived class is sliced off from the tuple, until the index reaches zero and the current value is the search for value. The value is then returned. The function “get” was declared with the inline keyword to give the compiler a hint that it should inline the functions. The full recursion is performed at compile time and therefore allows some optimizations to be done by the compiler. Looping over the complete tuple and applying a function can be done creating a class Loop which contains a function called apply (Code 2.4).

The class has three different template arguments passed to it. Those are the current and end index and the variadic template. The tuple is passed as argument to the apply function. An additional class which contains the function that should be applied must be created additionally. The apply function applies the function passing it the tuple element of the current index, using the “get” function. Then the apply function is called again with an incremented current. This is done until the current index is equal to the end index. This way it is possible to apply different functions on the tuple elements and to iterate over the elements in different ways. A static assert can be used to prevent out of bounds problems.

2.2.4. Tile-based Algorithms

To perform efficient inference and training of NNs the operations used must be efficiently implemented, the most important of them being the matrix multiplication and the convolution operation. The reason for this is that for example activation functions are in general only linear in the number of neurons while the matrix multiplication and convolution both have a higher time complexity also, the number of neurons is often relatively big. Both can benefit from one general approach used for implementing high-performance GPUs which is called tilling [64]. Tilling splits the result of the kernel into multiple tiles where each work-group is used to compute one tile. It is a natural approach in the case of a work-group model. It is especially useful when adjacent parts of the output make use of adjacent parts of the input because this can then allow efficient memory access. This is especially true when this allows coalesced memory access because memory that is used by many threads can be read with only a few transactions therefore increasing the ration of memory accesses to instructions. This ratio should be as high as possible because accesses to memory take much longer compared to instruction. Too many memory accesses can possible result in an underutilization of the GPU [61]. This approach often allows to make efficient use of the shared memory because the global memory can be loaded in a tile-wise fashion as is the case for matrix multiplications. It also allows the in-place transformation of data because data that would normally be transformed into another style, before the real computation can be performed in the local memory and therefore don't need to be performed in the global memory if the intermediate result needs not necessarily to be saved.

2.2.5. Image to Column

While convolutions can be implemented in a direct way using tiling an alternative approach is to transform the input of the convolution in a way that allows the convolution to be performed as a normal matrix multiplication therefore allowing the use of highly

optimized matrix multiplication implementations [65]. Matrix multiplication is on the GPU very efficient because it has a high rate of operations per read instruction [64] therefore, a transformation may be very useful especially if the resulting matrices are big because matrix multiplication is especially efficient for big matrices. The dimensions of the created matrix image are the following.

$$h_{mat} = w_k h_k d_k \tag{2.55}$$

$$w_{mat} = n_{pos} n_{batch} \tag{2.56}$$

Here w_k is the width of the kernel, h_k the height of the kernel, d_k the depth of the kernel, n_{batch} the number of images in the batch and n_{pos} is the number of kernel positions in the image which, can be calculated in the following way:

$$n_{pos} = n_w n_h = \left\lceil \frac{w_i - w_k + 1 + 2p_w}{s_w} \right\rceil \left\lceil \frac{h_i - h_k + 1 + 2p_h}{s_h} \right\rceil \tag{2.57}$$

n_h, n_w are the number of position in the height, width of the image respectively, s_w, s_h

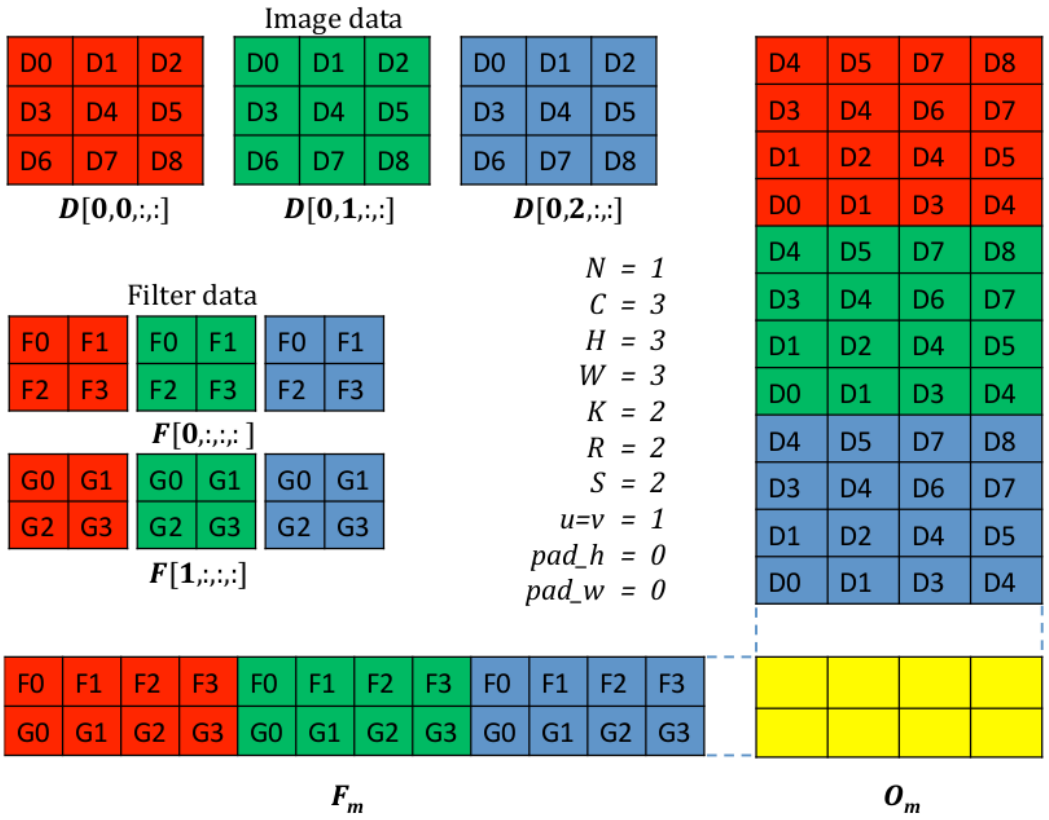

Figure 2.13 Image to Column Example [66]

are the stride in the width, height direction and p_w, p_h are the padding of the convolution in the width, height direction. An example on how the elements are structured before and

after the transformation can be seen in Figure 2.13. The filter map consists of many filters that compute the result of one feature map. Those single filter map elements represent the rows of the filter matrix which, has as many rows as there are output feature maps as can be seen in the Figure. The rows of the filter matrix are compromised of the unrolled filters for each dimension in the input depth. The filter can essentially be saved all the time this way in memory therefore making no transformation necessary. The input feature map matrix construction is a little bit complicated. It has as much columns as there are filter positions in the input image. Each column consists of all pixels that are in the respective filter when it is at the respective position. It starts with the unrolled first image pixels followed by the unrolled second image pixels and so on. One benefit of this approach is that in deep learning following product often stays relatively large [66]:

$$d_i n_{pos} \tag{2.58}$$

The first part of the product determines the number of rows of the filter matrix while the second part of the product determines the number of columns of the image matrix. Both of these dimensions represent the dimensions of the matrices that can be executed in parallel and are therefore a metric of how many operations can be performed in parallel. The possible parallelization archived by this approach is therefore relatively constant over every layer of the neural network and therefore also the performance of this approach. One disadvantage of this approach is the fact that the data must be stored duplicated therefore increasing the overall memory requirement of the algorithm. This is a problem because the memory requirements of deep learning algorithms, especially during training is performed, are quite high even without such an approach therefore making it possibly impractical if so much additional memory is needed. An alleviation of this requirement is the creation of the matrices in the local memory in a tile wise fashion as was proposed in [66]. There is not much information on how this was done exactly in the paper so a custom implementation of this approach was implement and will be explained in a later section.

3. Related Work

There already exist a few different deep learning frameworks which can be used to create different neural networks and train them on the GPU. The most popular frameworks are Caffe [67], Theano [68], Thorch [69] and most recently Tensorflow [70]. All the frameworks have different advantages and disadvantages. They allow the use of a high-level language like Lua or Python which allows fast and easy creation of different models. Caffe for example is very good for CNNs for which it was originally developed. It is

relatively inflexible because the neural network is created layer wise and if one layer is not defined, one must adapt the source code. It provides a rich collection of different models called the model zoo. Torch is comparable to MATLAB, in how models are created, and allows the creation of many different neural network architectures. Theano also allows the creation of many different neural networks, but instead of only running the neural network it also compiles the NN, that is created by the user. Tensorflow also allows fast and easy creation of many different architectures and it also supports easy distributed training on many devices without bothering the user with all the details. All four frameworks make use of cuDNN to implement the operations on the GPU device. The support of OpenCL is fairly restricted at the moment. For Caffe exists an experimental OpenCL framework, that is maintained by the community [71]. Torch has multiple different OpenCL backend implementations [72], but none of them are part of the native implementation. For Theano an OpenCL backend implementation is in work but the support is incomplete [73]. This shows, that OpenCL is not widely supported in the deep learning community, the support is increasing.

An important part of the deep learning framework is how the specific operations are implemented, especially the convolution operation. There exist different options on how the operations can be implemented. One implementation is to make use of the fast Fourier transformation to transform the image and kernel into the Fourier space, perform an element wise product there and then transform it back with the inverse fast Fourier transform [74]. While this algorithm is relatively fast the draw-back is the memory requirements because the transformed filters must have the same size as the input image. It increases the size of the parameters that are stored, because typical filters for deep learning are rather small [66], and the fast Fourier transformation is for small filters not as efficient. On the other hand, it could be useful for the backward pass because in this case, a convolution between the input image and the gradient of the output must be performed. An alternative algorithm is to transform the convolution to a matrix multiplication [65] which can be performed tile wise in memory [66].

Deep learning was used in a few works to solve the problem of handwritten recognition [45] for example used so called multidimensional LSTMs(MDLSTMs) and the CTC loss to perform handwritten character recognition. They reached good performance and in [53] this approach was expanded by incorporating attention. This addition allowed the application to whole paragraphs. While the application of MDLSTMs achieves good results, they have the disadvantage that they are relatively sequential which makes them less useful on the GPU. Making use of convolutional layers would have the benefit that

there exists an approach which allows the reduction of total parameters while keeping the overall accuracy of the system [75]. This would enable the use of this system by more memory restricted devices like FPGAs and embedded systems. CNNs where successfully applied to generating captions for images in [52] making also use of attention. Even though, the problem of generating a caption for an image is different from reading text contained in an image, both are somewhat related. In both cases the goal is to decode the content of an image into text. [76], [77] used CNNs to successfully read the text in natural images. In [78] LSTMs were used to read historic handwritten Latin text by feeding only one pixel wide columns of the image to the LSTM at each time step.

4. Implementation of the Framework

This chapter begins by giving an explanation on how the framework was implemented in C++ and OpenCL. This is explanation will point out the most important parts and the problems that were to be solved. Then three different neural networks, which are used to perform handwritten character recognition, will be explained followed by the details of the experiments that were conducted using them.

4.1. The Framework

The framework was developed having three requirements, the system should fulfil. Those are a reasonable performance, extensibility, and cross-platform compatibility with respect to the used hardware. The system should be high-performant because the systems goal is to train Neural Networks and deploy them. The performance of the system has a major impact on the usability, because the training could take too long. Inference done by the framework on new incoming data needs also to be reasonably fast for it to be useful. For those reasons, OpenCL was used to implement the basic algorithms which are common to many Neural Networks.

Extensibility is needed because the field of deep learning is rapidly progressing and new layers/architectures are discovered steadily. Those new layers can then easily be implemented and fast used if the framework was developed in an appropriate way. Also, because it is essentially a framework for deep learning it can easily be adapted to new problem settings by adding a few new operations.

The last requirement on the developed framework is the flexibility regarding the useable hardware. It should be possible to use the framework on different hardware devices like CPUs, GPUs and FPGAs and make use of the available parallelization. For this reason, we decided to use the OpenCL API and don’t make use of CUDA, which is only able to use GPUs from NVIDIA. Though CUDA gives the programmer more options to optimize

its code because it gives access to more hardware specific operations. OpenCL needs to keep its cross-platform compatibility. Caused by this it gives not natively access to those operations. Also, OpenCL 1.2 was used and not a newer version of OpenCL, because NVIDIA GPUs only support 1.2 and no newer versions. The deep learning framework consist of four different parts (Figure 4.1) which are, a collection of OpenCL kernel, the Backend system, the Neural Network system, and the Data system.

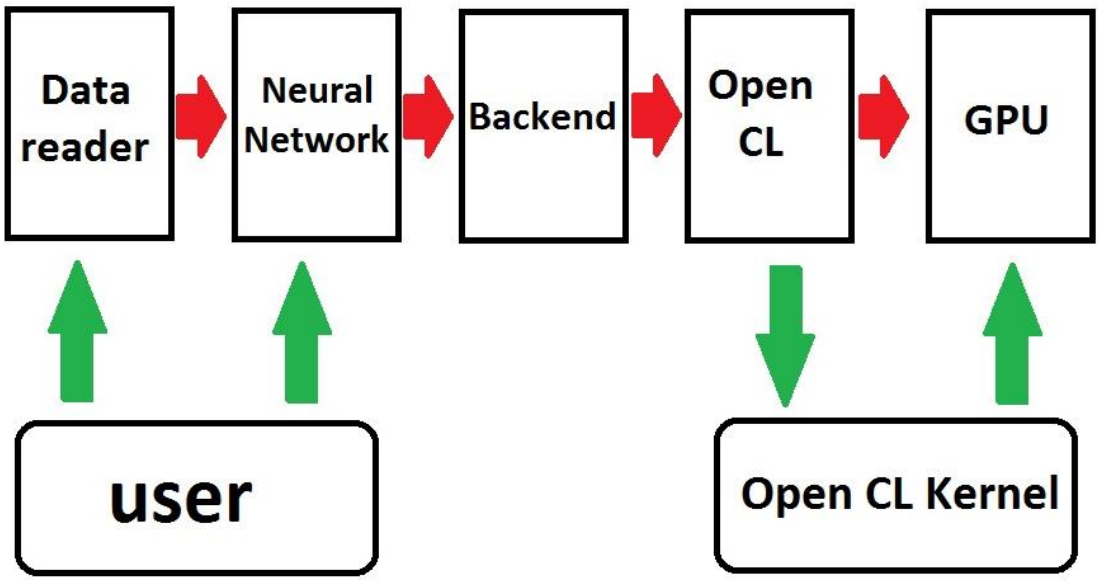

Figure 4.1 Framework Architecture

The framework represents the neural network as computation graph, where nodes represent basic operations. The edges represent the data, called buffer, that is passed from one operation to another operation or rather data that is saved in the system. The user of the framework uses the neural network sub component to build a computation graph out of basic operations. While creation of the neural network no data is allocated on the devices or specific operations are conducted. Only “abstract” operation and buffer objects are constructed to represent the computation graph. They only represent operations and data but contain no code for operations or data. The computation graph needs then to be explicitly initialized before it can be used for training or inference. At this stage, advanced analysis of the graph can be done even though none is implemented at the moment. For the initialization of the computation graph is the Backend used to create concrete operation and buffer objects in form of kernels and buffer objects in OpenCL. The operations in the backend will be called backend operations or hardware operations and the buffer objects in the backend will be called backend or hardware buffers. All system components use indices to point on buffer objects and operations in the different sub systems. This allows a decoupling of the underlying implementation details from the rest

of the system. The backend is the only system that accesses OpenCL in any way because it abstracts all OpenCL specific information away, it gives no access to the underlying OpenCL object. Because OpenCL is completely abstracted away from the rest of the system it is possible to easily swap out OpenCL by another API. It is only necessary to change the backend system while the Neural Network and Data system can stay the same. The task of the backend is to make decision regarding the configuration of OpenCL and the kernels that are used. The basic operations that run on the hardware are given by OpenCL kernels and are contained in the collection of basic kernels. It contains kernels for basic operations like matrix multiplications, convolutions, and different activation functions. The remaining system is the data system which task it is to handle the loading and pre-processing of data. It can load data asynchronous from the hard drive from one or multiple files. It then pre-processes the loaded data, for example it can pad images or time-series data to a specific size. The details of all four sub-systems will now be given starting with the Neural Network system followed by the backend. Afterwards the OpenCL kernel collection will be explained and then the Data system.

4.1.1. Neural Network Sub-System

The NN sub-system is the main system with which the user interacts. It is used to create the model, save the model, and control the training and inference. It itself consist of

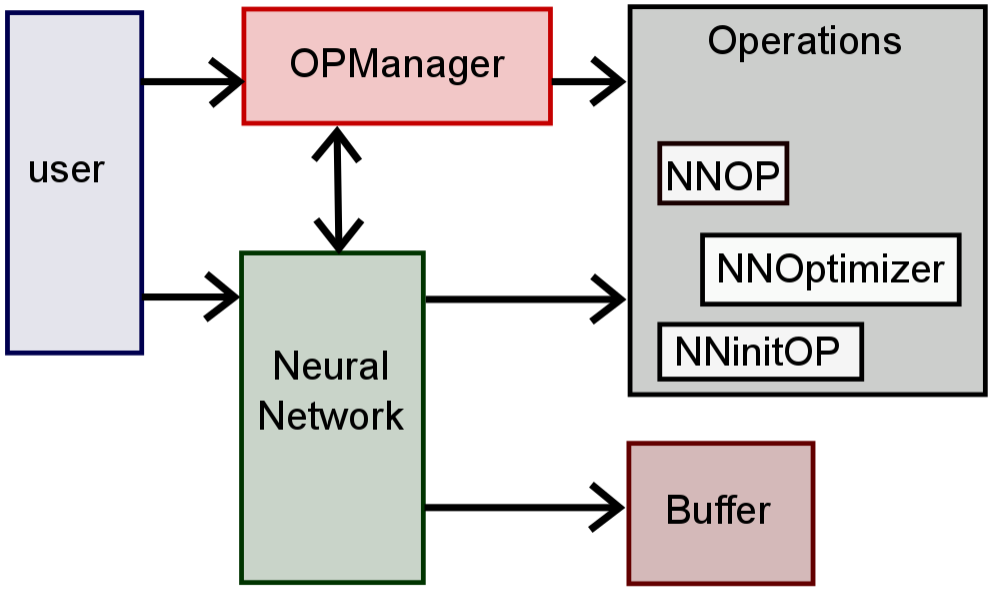

Figure 4.2 NeuralNetwork System Architecture

multiple classes, which are the NeuralNetwork, NNOp and its derived classes, NNBuffer and its derived classes, NNOptimizer and its derived classes, InitOp and its derived classes, SizeVec and the OPManager. A diagram of the NeuralNetwork system architecture can be seen in Figure 4.2. The system comprises of three different thematical components. The main component is the Neural Network class which contains all data relevant to the neural network building process. The operation part of the system consists of all parts relevant to the creation and specifics of operations, which are all classes except the NeuralNetwork class and the buffer classes. It contains the details of the different operation types and provides functions to ease the create of those operations. The last component are the buffer classes which store information about the data that is used in the NN and stored for each operation. The goal behind the Neural Network sub-system was to provide an easy and direct interface for the creation of different NN architectures and the associated training of it. To understand why the NN sub-system was build the way it was one must look at the pipeline of deep learning. At the start of the pipeline stands the creation of the deep learning model this includes the conception of the model that is to be used but also the implementation of it in software. The fastest way to create such a model is to make use of different framework as the one written in this work. After the model is created one must initialize the parameters and the system that is used for deep learning. After the complete model is defined and initialized the training can start. For the training, one must retrieve batches of data from the dataset that is used to train the model. After the model is finally trained the model can be used for inference in application setting. All steps in the pipeline underlie different restrictions. While the creation of the model by the user does not really need to be very performant, because in general it does not require much calculations, creating the model should be straight forward because the user should be able to focus on the details of the model and how to implement it. The system should therefore be expressive enough to build a wider variety of models while keeping it straightforward. The initialization of the model and the system can take a relatively long compared to the operations in the model creation. The reason for this is a trade-off between training/inference time and initialization time. If more time is spent in the initialization of the model and analysis of it with the goal to reduce the overall training time one can safely make use of this trade-off because the time initialization takes is in general magnitudes lower than the time training takes. A more time expensive initialization for inference may also be very beneficial if the time taken for inference is reduced and therefore more inferences in the same amount of time can be

done. Therefore, while the NN sub-system should not be too slow it does not really underlie time constraints as for example the Backend does.

An example of how a Neural Network can be implemented and trained using the framework can be seen in Code 4.1. It shows the creation of a linear model with a squared loss, the Adam algorithm as optimizer and the initialization using a normal distribution. After the model is created it shows how training can be performed. At first the user creates a NeuralNetwork class instance, this instance will then be the default NeuralNetwork. It is possible to create multiple NeuralNetwork instances at the same time but the first is always the default NeuralNetwork to which all operations will be added. The user can then create variables and operations which perform some computations on them. The operations are created by using the OPManager class and they are added to the current default NeuralNetwork. If the user wants to add operations to a different graph one must set this graph explicitly to active before it is possible to add operations to this graph. In the displayed case two input variables and a parameter variable are created. The input variables are used to input the data and labels into the computation graph while the parameter variable represent the trainable parameters of the model. The Multiply operation then conducts a matrix multiplication on the variables and returns a new intermediate variable. At the end, a loss is added which is in this case the squared loss. An optimizer also needs to be added to the NeuralNetwork class. The trainable variables are initialized by adding initialization operations. Different methods for initializing the parameters are implemented, examples are the normal, uniform and truncated normal distributions. After the creation is done the graph needs to be initalized. This is done by calling the NeuralNetwork member function InitalizeGraph. This function initializes all operations and data buffer and can potentially analyze the graph. As parameter is the batch size, that is used for training, passed to it. After the model creation and initialization is done the user can call Forward and Backward. Forward is used to run the forward path of the computation graph and Backward is used to compute the complete backward path. In the example, the Forward functions receives pointers on data that is then transferred onto the GPU into the input variables. The Backward function always computes the gradients with respect to the last forward path so forward must be called before backward can be called for the first time. Backward does not update the parameters of the model this is done after Backward by calling BatchDone on the NeuralNetwork instance. This operation updates the parameters using the specified optimizer using the calculated gradients and then does some updates and clean ups. In the example Forward, Backward

and BatchDone is called 1000 times in a for-loop, the model is therefore trained for 1000 steps.

```
	//Create and Init NN
	DeepCL::NNSystem::NeuralNetwork nnTest;
	nnTest.InitSystem();

	//Build Model
	DeepCL::NNBufferIdx i = nnTest.CreateInputBuffer(784);
	DeepCL::NNBufferIdx label = nnTest.CreateInputBuffer(10);
	DeepCL::NNBufferIdx wf1 = nnTest.CreateParameterBuffer(784, 10);
	DeepCL::NNBufferIdx bf1 = nnTest.CreateParameterBuffer(10);
	DeepCL::NNBufferIdx linearRes = DeepCL::OP::Multiply(i, wf1);
	DeepCL::NNBufferIdx linearResBias = DeepCL::OP::AddBias(linearRes,
bf1);
	DeepCL::NNBufferIdx   loss   =   DeepCL::OP::SquaredError(linearResBias,
label);
nnTest.AddOptimizer(new DeepCL::NNSystem::NNAdam(0.0001f, 0.9f, 0.999f, 10e-
8f));

	//Add Initalization operations
	DeepCL::OP::InitWeightNormalRnd(wf1, 0.0f, 0.1f);
	DeepCL::OP::InitWeightUniform(bf1, 0.f);

	nnTest.InitliazeGraph(5);

	//Create Data Loader in this case for the MNIST images
	DeepCL::DataSystem::IDXReader idxReader("train-images.idx3-ubyte",
"train-labels.idx1-ubyte", BATCH_SIZE);
	DeepCL::DataSystem::MNISTTransformer idxTransformer(BATCH_SIZE);
	DeepCL::DataSystem::BatchManager<2, int, float>
batchManager(BATCH_SIZE, 1, 25, &idxReader, &idxTransformer);

	//Start Training
	for (int c = 0; c <= 1000; ++c)
	{
		DeepCL::DataSystem::Batch<int, float>* batch =
batchManager.GetBatch();

		//The function expecst point on the data and not std::vectors.
		nnTest.Forward<int, float>(DeepCL::BackendSystem::get<0>(batch-
>data).data(), label, DeepCL::BackendSystem::get<1>(batch->data).data(), i,
batch->sizes[0], batch->sizes[1], batch->batchSize);
		nnTest.Backward();

		nnTest.BatchDone();
	}
```

Code 4.1 Framework Example Use

Since the workflow was explained in some detail the different components will be explained in more detail starting with the computation graph that represents the NN. The graph consists of derived class objects of NNOp and NNBuffer, where NNOp and NNBuffer are abstract classes. The NNOp represents the operations that are performed by the Neural Network while the NNBuffer represent all the data that is handled in the

NN, including intermediate results, input data and trainable parameters. Those objects are stored in the NeuralNetwork class which contains two different dynamic arrays. The first one, nnOperationList, stores the operations in form of pointer on NNOps objects and the second, nnBufferList, stores the buffer in form of pointer on NNBuffer objects. This is the only place where real reference/pointers on the graph elements are stored. All other classes that use NNBuffer and NNOp objects only store indices into those arrays which are called NNBufferIdx for NNBuffers and NNOperationIdx for NNOps. This way bounding checks can be made and the risk of pointer pointing onto destroyed objects will be reduced. A more important benefit of this approach is that only indices are returned to the user. The user can only access the buffer information and the operation information through the NeuralNetwork object, this allows the NeuralNetwork to abstract away the details of the underlying system. The graph elements are stored in an array and not in another data-structure because arrays allow cache efficient data use. An alternative implementation would be lists but lists are not as cache friendly if one needs to iterate over them. Iterating over the elements is important when the graph elements construct the real operations which then later run on the GPU. To represent the array as graph or to be more precise as directed graph the NNOp objects save the indices of the in-going and out-going buffers. The NNBuffer objects store the indices of the operation from which they are a result and the operations in which they are used. Normally a graph edge only has one target, which is still somewhat true in this case because, when a buffer is used multiple times it can be thought of as being duplicated. This is never done explicit because it would increase the total number of buffers and therefore the memory requirements and additionally it would be more inefficient. Since NNBuffer objects are used to represent real data that is used in the NN, it needs to store information about the size of the object. The size is split up into two different parts, which are a SizeVec object and the time size. The SizeVec is only used to store four sizes as members and allows an easier passing of sizes to functions and so on. The SizeVec is four-dimensional because many different objects in the deep learning framework can be best represented by four dimensions, an example is a batch of images where the first dimensions represent the batch size, the second the color depth, the third the height and the last one the width. Another one is the parameter of a convolution where the first dimension represents the number of filters (depth of the output volume), the second represents the depth of each filter (depth of the input volume) and the last to represent the height and width of the filter. The time size represents the length of sequences and is handled differently because it is assumed that different time steps have different associated buffers on the hardware. The buffer objects

in the NeuralNetwork class contain no real data, because the data is stored and allocated in the Backend sub-system, even so the buffer needs to have knowledge about its associated hardware buffer. Also since there are two passes through the NN, each buffer needs to have a forward and a backward buffer which stores the gradients. For this purpose, the NNBuffer class contains four different member variables. The first two variables are for the forward pass which are an index on a hardware buffer in the OpenCLBackend and a dynamic array of such indices. The array of indices stores indices of hardware buffers for each time step. The single index points onto a hardware buffer that contains all time step hardware buffers. This is necessary to have a hardware buffer, which can be used to iterate over all time steps in a kernel. The second two member variables are equivalent but for the backward results. It also contains a variable for the current time step which is necessary for the creation of the operations. The different indices can be accessed in different ways with a few member functions. It contains four different virtual functions which are necessary to specialize the buffers, that inherit from it, on different use cases. Those are ForwardBuffer, Instantiate, SetBatchSize and Reset. ForwardBuffer is used to return the forward buffer of a specific time step. Instantiate is used to create necessary Backend buffer via an OpenCLBackend instance. This way the buffer only needs to know what number of hardware buffers are necessary and not the NeuralNetwork class. The SetBatchSize function is used to automatically set the last parameter to the size of the batches if necessary. Reset is used to reset the buffer objects to a default value if necessary. There exist five different derived class which are NNInputBuffer, NNIntBuffer, NNParamBuffer, NNStateBuffer and NNTmpBuffer. The NNInputBuffer, NNParamBuffer and NNStateBuffer can be created directly by the user. The NNInputBuffer allows the passing of data to the GPU, the NNParamBuffer contains trainable parameters and the state represents the state of a RNN layer. NNInBuffer objects are generated as a result of an operation and the NNTmpBuffer is used for temporal results which only need to be stored for this operation. The matrix multiplication operation for example needs a temporary buffer because in the gradient computation one matrix must be transposed before the correct matrix multiplication can take place. This transposed result is stored in such a temporary buffer.

The operations are created using the OPManager class, which serves as factory for the creation of different Operations. When a NeuralNetwork class object is created for the first time, a static pointer onto an NeuralNetwork is set to it. This way the first created NeuralNetwork object is always the default object to which the operations are added. An NeuralNetwork objects can be set as active by calling the static SetActiveNN function

passing an NeuralNetwork pointer to it. The class contains static functions for many operations which can be added to the computation graph. When an operation is created, the OPManager creates the derived NNOp object and adds it to the computation graph of the active NeuralNetwork object. The NNOp class is the base class for many different operations that are implemented in the framework. It stores all relevant information and

```
void NNReLUOp::Instantiate(BackendSystem::OpenCLBackend& backend,
std::vector<NNBuffer*>& bufferList)
		{
			const int WORK_GROUP_SIZE_X = 32;

			//Retrieve the Buffer
			NNBuffer bufferA = *bufferList[input[0]];
			NNBuffer bufferC = *bufferList[output[0]];

			//Calculate linearized size
			size_t totalSize = bufferA.size.sizeX *
bufferA.size.sizeY * bufferA.size.sizeZ * bufferA.size.sizeW;

			//Create the Tuples for the forward and backward
operation
			Tuple<BufferIdx, BufferIdx, dataPair>
tuple(bufferA.ForwardBuffer(), bufferC.ForwardBuffer(),
				dataPair(sizeof(int), totalSize));
			Tuple<BufferIdx, BufferIdx, BufferIdx, dataPair>
tupleGrad(bufferA.ForwardBuffer(), bufferC.BackwardBuffer(),
bufferA.BackwardBuffer(),
				dataPair(sizeof(int), totalSize));

			//Query the Kernel indices
			KernelIdx reluKernel = backend.GetKernelIdx("ReLU");
			KernelIdx reluKernelGrad =
backend.GetKernelIdx("ReLUGrad");

			//Add the Operation to the Backend.
			OperationIdx  matOp = backend.AddOperation<3, BufferIdx,
BufferIdx, dataPair>(reluKernel, tuple, cl::NullRange, cl::NDRange((totalSize
+ WORK_GROUP_SIZE_X - (totalSize%WORK_GROUP_SIZE_X))),
cl::NDRange(WORK_GROUP_SIZE_X),
BackendSystem::OpenCLBackend::OperationType::FORWARD);
			forwardOpIdx.push_back(matOp);
			matOp = backend.AddOperation<4, BufferIdx, BufferIdx,
BufferIdx, dataPair>(reluKernelGrad, tupleGrad, cl::NullRange,
cl::NDRange((totalSize + (WORK_GROUP_SIZE_X -
(totalSize%WORK_GROUP_SIZE_X)%WORK_GROUP_SIZE_X))),
cl::NDRange(WORK_GROUP_SIZE_X),
BackendSystem::OpenCLBackend::OperationType::BACKWARD);
			backwardOpIdx.push_back(matOp);
		}
```

Code 4.2 ReLU Instantiation Host

functions to allow the operation to create its necessary operations in the backend. The member variables of the class are an operation which are two dynamic arrays for indices of backend operations, the indices onto the buffers that are the in- and output of the

operation, two dynamic arrays for the temporary buffers and two parameters for the time step. The two-time parameters are used to control how the NN behaves over multiple time steps. The first parameter contains the time offset at which the operation is performed for the first time. The second parameter contains the number of time steps this operation will run. The parameters allow operations to be executed later step which is useful for operations like when a class must only be returned at the end of the sequence. The two arrays that contain the backend operations store indices of operations that will later be performed on the GPU, because this class only represents an abstraction of operations since no code that is executed by an operation is available in this class. The real operations are implemented in the GPU operation objects. One array stores the operations that are executed in the forward pass and the other one stores the operations that are calculated by the backward pass. There may be more than one backend operation for one NeuralNetwork operation because some gradient calculations need multiple simple operations like the gradient of a matrix multiplication which needs to transpose both inputs once and a matrix multiplication for both. The first array of the temporary buffer stores the indices onto hardware buffer objects which are used for intermediate results. The other array contains the size that is needed for each temporary buffer. The two are dynamical arrays because different operations may make a different number of temporary buffers necessary. Most operations don't need any temporary buffers. The NNOp class contains 4 virtual functions, those are Instantiate, GetOutputType, GetTimeTransform, SetTmpBuffer. The SetBufferTmpBuffer is used to set the required sizes for the temporary buffers when the operation is fully specified. The GetOutputType function is used to transform the input sizes into a correct output size. This way only the derived class needs to know how the resulting size is computed. The GetTimeTransform does the same as the GetOutputType function but this time calculating the correct output time variables. This is necessary because there are operations that take an input which is of time length one but it returns a sequence which is longer. The Instantiate function is the most important of those and different for each derived class. Calling this function creates the backend operations necessary for computing the forward and backward pass of this operation. A OpenCLBackend object reference and the array of NNBuffers from the NeuralNetwork object is passed to it. This is necessary because the object needs to have access to the backend to create the operations in the backend and to get access to the real NNBuffer objects since it has only indices stored. The direct access at the NNBuffer object array is acceptable because the operations are part of the Neural Network Sub-

System and it needed to have access to the information in them. The alternative would have been to query the NeuralNetwork for each but this would have been more inefficient. Code 4.2 displays the implementation of the ReLU operation. Each derived Operation stores additional information necessary for the operation, for example the stride of the convolution.

It starts by storing the buffer objects in local variables. It then creates two tuple objects which store the function arguments for the forward and backward pass operations. The specific hardware forward and backward buffers are queried by calling ForwardBuffer/BackwardBuffer on the respective buffer object. The functions return the hardware buffer of the current time step allowing the operation to be instantiate without it having to care about the current time step, for which it is instantiated, and in what way the Buffer provides hardware buffers (intermediate buffer can return a hardware buffer for each time step while the parameter buffers return the same hardware buffer for each time step). It then queries the backend for the indices of the “ReLU” kernel and the “ReLUGrad” kernel by name. Then a forward and backward and operation is created and passed to the backend object, while signaling it which pass this operation belongs to. Operations that need more backend operations create more tuple objects, create more backend operations and so on.

The initialization operations are not derived from the NNOp but from the IniOp class, because it has different needs. It needs for example member variables for random number generation and it also acts only on one buffer. It again has an Instantiate function but it only creates the data, in a way depending the derived class, using for example a normal distribution, and it then writes it into the OpenCL buffer object via the backend object.

The Optimizer operations are also handled separately and are not derived from the NNOp class. They derive from an NNOptimizer class which contains two different virtual functions. Those are GetNumBuffer and Instantiate. The GetNumBuffer function returns the number of additional buffers necessary for things like momentum. The instantiate function is called ones for each trainable weight parameter and creates a backend operation for each one. The backend operation adds the gradient to the current weight and updates possible momentum terms therefore the it is only called when the update pass of the OpenCLBackend is performed

The graph of operations and buffers is stored in the NeuralNetwork class, therefore it contains different functions for executing and managing it. Among others it contains functions for creating input, state and parameter buffers and also for adding different operations. The functions for adding operations are used by the OPManager and they

return a result buffer for it if no result was passed to it. It also functions for adding optimizer operations and weight initializer operations to the graph. Only one optimizer can be used at the same time so it is directly stored in a pointer. To create the resulting buffer, it calls GetOutputTime and GetTimeTransform of the operation to create a buffer of the correct sizes. It provides functions for reading, writing and printing into some output stream of different buffers. It allows also to store the model in a file and load from it. Only the parameter buffers are stored and restored when the model is saved, therefore making the code for creating the model still necessary. The parameters are stored in the file using a mapping from index to name, therefore each parameter needs to be stored with a different name. The most important functions of the class are the Forward, Backward, BatchDone and the InitalizeGraph function with all the functions it calls. The Backward and BatchDone functions exists only in one version each which calls the run function of the backend with the respective OperationType constant BACKWARD or UPDATE. The BatchDone additionally calls the ClearBackwardBuffer function which sets all backward buffers to zero. The reason for this is that the gradients are always added to the backward buffers because no explicit duplication operations are performed. The overhead of doing this is not as much but it can be further optimized by analyzing the graph on implicit duplication operations and then choosing the appropriated kernel respectively. This approach would still make it necessary that the duplicated buffers are reset after each update but the number of buffers would be reduced. For the Forward function exist different implementation about the type and number of parameters they take. The most basic version only calls Run of the backend with the FORWARD operation type constant. Other versions get a different number of input buffer indices and pointer onto data passed to which is then written into the input buffers before calling run. The most elaborated version allows the passing of an arbitrary number of dynamic arrays and indices to it. This way it is possible to have an arbitrary number of input buffers. The variadic template argument is unrolled and all elements in it are written in the respective buffers. Forward, Backward and BatchDone can only be executed when the graph was initialized using the InitalizeGraph function. The InitalizeGraph function is used to initialize all important parts of the graph and to create the necessary OpenCL resources using the backend. It starts by calling the SetBatchSize function on each buffer, passing it the batch size used for this run. This allows each buffer to set the batch size component in the size to the correct value if necessary. The parameter buffers don’t change anything because they are independent of the batch size. Afterwards the CreateTmpBuffer function is called, which calculates the number of temporary buffers necessary and the size of each

to minimize the number and size of each. The buffers are then created and the indices are passed to the operations whom need them. Since now every necessary buffer is known each one is instantiated by a call to InstantiateBuffer. This function calls Instantiate on each buffer, allowing the buffer to create its necessary hardware buffer using the backend. The same is then done with the operations, with a call to InstantiateOperations, since now all backend buffers exist. This order is necessary because the backend operations need the indices of the backend buffers making it necessary that each buffer was already instantiate. This functions calls Instantiate of each operation which allows the operation to create the necessary hardware operations for each pass. The Instantiate function is not called once but multiple times for each time step the operation is called and only if the current time step for which the operation is created is bigger than the offset. The reason for this approach is that each operation at each time step only depends on the current time step and all before. This means that representing an RNN as unrolled graph the operations following the depth are created first before the next time step is created. The same is true for the backward pass but this time in the opposite direction. The graph is essentially created once for each time step. Furthermore, iterating through the array of operations is okay, because an operation using the results of another operation can only be created, when the result of the other operation is available. The other operation must be created before this one and has therefore a position before the deeper operation, that used the result, in the array. After each iteration through the operation array the current time step in each buffer is incremented. This way each iteration through the array will use a hardware buffer of another time step, if the derived buffer object behaves this way. The parameter buffers for example always provide the same hardware buffer for each time step. After this is done all buffers are set to zero followed by the initialization of each weight buffer through a call to InitalizeWeights. This function calls the Instantiate function on each InitOp. After everything is done the graphInitalized Boolean is set to true to allow calls to forward, etc. to be executed.

4.1.2. Backend Sub-System

The Backend is the only sub-system that interacts directly with OpenCL, all other sub-systems interact with OpenCL indirectly via the Backend. It is used to read, create and manage OpenCL kernel objects, devices and the memory buffer objects that are created via OpenCL. It consists of four different parts, which are all part of the BackendSystem namespace. It can be seen in Figure 4.3. The components are the OpenCLBackend class

a collection of operation classes, the tuple class and associated utility objects, and the KernelLoader. The main component of the Backend is the OpenCLBackend class which serves multiple purposes. One purpose is the initialization of OpenCL and the

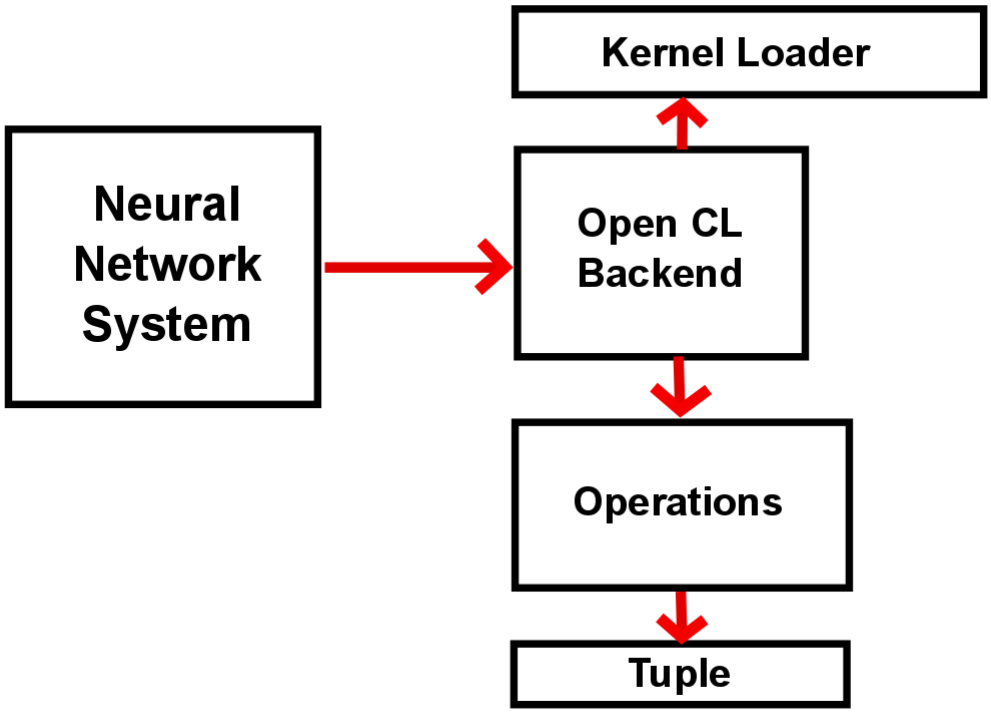

Figure 4.3 Backend Architecture

components needed by OpenCL. It is also used to create OpenCL buffer and OpenCL kernels dynamically as they are needed at runtime. Its last function is to provide the implementation of the different passes that are created by the NeuralNetwork class. The KernelLoader class is used to load all kernels from multiple files either by using a defined configuration file that list all kernel files that need to be loaded or the kernel files can be read one by one. The Operation classes represent the wrappers of operations that will run on the GPU and the Tuple classes purpose is to make passing arguments to the kernel objects easier.

Different OpenCL kernel need a different number and different types of parameters which raises the question on how to represent those parameters in the CPU host code and how to pass them to the GPU. One possible approach would be to create struct templates which derive from one base struct. Then for each number of parameters a custom struct must be created which can become quite cumbersome because the number of parameters can range from only three for a ReLU kernel to eleven arguments for a convolution kernel. An alternative approach is to store the kernel arguments in tuple objects and then to loop

over those objects when they arguments need to be set. The decision fell on Tuple object since they allow the passing of arguments in a flexible and type save manner. They are handled in the operation classes which are used to execute OpenCL operations. They store all operation relevant information like the tuple of parameters, the OpenCL kernel to be executed and the work group sizes, for the OpenCL kernel. The BaseOperation class only contains an abstract virtual function Run which is used when the operation should run. There exist two different derived classes, Operation and IncrementOperation. Operation is used to execute an OpenCL kernel (enqueue it in the queue) when the Run function is called. It loops over all tuple arguments and sets them using the OpenCL API. For different tuple element types exists different implementations of the function that is applied. One is used for an OpenCL buffer object that needs to be set. Parameters like normal integers must be as pairs to the tuple, where the first element of the pair contains the size in bytes of the data object and the second the data object. After every tuple element was set, the kernel is enqueued in the default queue. The queue executes all operations in order. The run function of the operations must therefore be executed in the correct order. The IncrementOperations increments an element in the tuple by one after the kernel was enqueued. This is necessary when the current number of execution is imported, as is the case for the Adam optimizer.

The third part of the Backend sub-system is the KernelLoader class which consist of two different static functions. The functions are only used for reading kernel files in ASCII format and they don’t need to have any state which is the reason why both functions are static. The first function called LoadKernel gets a file name passed to it and returns a dynamic allocated string containing the data read out of the file. The second function LoadKernelConfig gets the file name of a configuration file passed to it and returns a pointer on a dynamically created dynamic array of strings which contains the path of all files named in the configuration file. Each line of the configuration file has either a name or an absolute path of a file containing one or multiple kernels written in it. If it is only the name of a file, the file must be in the same folder as the configuration file. LoadKernelConfig then reads all files into strings and pushes them in a dynamically allocated array. Using a configuration file allows easy addition or replacement of kernel files. All kernels are stored only as ASCII files and not as compiled binary files because the kernels are created dynamically at runtime for different compile time define arguments. This way the same kernel source can result in multiple different compiled kernel programs. The OpenCLBackend class ties all other components of the Backend sub-system together and additional connects the OpenCL kernel collection with the NN

sub-system. It is used to create and store OpenCL objects. It also controls the execution of the kernel objects and partially the order of the execution. Before it can be used two functions must be called after construction of the class. The first is InitGPU which initializes all OpenCL components needed to run the NeuralNetwork on different devices and the second is load kernel which loads the kernel objects and prepares them for further use. InitGPU retrieves information about the available devices and platforms, using the OpenCL API, and lets the user choose a device/platform to use. This determines the device on which the NN will run. After the user has chosen a device, more specific information of the device will be queried, which are necessary for the buffer and sub-buffer creation. It also creates the command queue of OpenCL, which can be done with or with the possibility to receive profiling information. Out of order execution of kernel is disabled because it is necessary that all operations are performed in the order they were enqueued because the kernel are strongly dependent on the order in which they are executed. All those informations are stored in members of the class.

The second function that must be called before the Backend can be used is called LoadKernel which loads the kernel config file and extracts the kernels source codes from the kernel files. The code for each kernel and the respective name are stored in a map data structure, with the name as key, to be able to retrieve the source code for each Kernel that is needed.

At creation of the framework an important decision had to be made. It was necessary to decide upon data layout, in memory, regarding the time-series dimension of the data. While the normal four-dimensional structure can easily be represented by a normal buffer object representing the data as flattened array, the time dimension of the data possesses very complex requirements that in some sense compete. The first requirement is that kernels should be independent of the time step at which they are applied to the data. This means that the data should be passed the right way to the kernel so that no adaptions depending on the time-step are needed. This requirement allows the use of some arbitrary kernel even if it was not created for the use with time series data. This is also important because in the case of inference the intermediate results must not be stored and therefore changing the way the buffers are used to a double buffering approach. The second requirement is that it must be possible to iterate over all time-series data objects in a kernel. For example, this is needed to calculate the average loss over all time steps. Iterating over them should be as directly as possible so that kernels that are more general can be used on this problem. One solution that fulfils both requirements is to create one big buffer where all time-steps lie behind each other in the memory, as is done for the

other dimensions. The kernel calls, at different time steps, would then get a pointer onto the respective buffer position. This way it would also be quite easy to iterate over the complete buffer but OpenCL doesn't support this type of approach directly because it is not possible to set the offset into the buffer when setting it as an argument for a kernel. One possible solution would be to set an additional kernel argument for the offset but this would contradict requirement one by requiring an adaption of the kernel to incorporate the offset. The second approach would be to use an NDRange offset corresponding to the time step but this would at least require an adaption since it changes the global index therefore needing again an additional adaption of the kernel. Also, one disadvantage of this approach is that it becomes problematic when kernel objects from two different time-steps are used. The approach that is used in the framework is to split a big buffer into multiple sub-buffer, for each time-step one. This essentially allows to set a buffer at a specific offset, but has the disadvantage that it requires the sub-buffers to adhere to some device specific memory alignment. This way requirement one is fulfilled while the second requirement not as much. The kernels that work on all time-steps need to get an offset value to index the different time-step values. But this disadvantage is acceptable because there are just a few kernels that need to perform such operations while there are significantly more kernels that need to operate on the normal data arrays. Another disadvantage is that it also takes a bit more memory because between the different time-steps unused memory is allocated. There exists another benefit of this approach because memory on the GPU should be accessed in coalesced manner, as was explained before, and the memory alignment requirement might help archive this. If the sub-buffer hadn't this requirement the sub-buffer might start at an unfortunate alignment so that, even if the kernel in principle loads memory in a coalesced manner, the memory access is not coalesced. There exist two functions, CreateBuffer and CreateSubBuffer, which are used to create buffer and sub-buffer. The CreateBuffer functions create buffer objects that can contain sub buffers for all time steps which adhere to the memory requirements. The CreateSubBuffer function creates a sub buffer at the correct offset. Both function can have a memory constant passed to them which specifies how the buffer is used (Read, Write or Read and Write). Both functions store the buffer in a list and they return only and index into the list to the caller. This way the system that calls these functions does not need to know the details of the underlying function and the way buffer are implement. This makes a change to the buffer implementation possible without the need to change anything in the other systems.

Not only buffer objects must be queried from the OpenCLBackend but also the kernel objects. For this reason, OpenCLBackend provides the function GetKernelIdx which returns a KernelIdx, used to index kernel objects in OpenCLBackend. There are two versions of this function the first only takes the kernel name as argument whereas the second also takes some define compile time arguments for the kernel. The compile time arguments must be in the form “arg1=value arg2 =value2 ...”, here arg1 and arg2 stand for the names of the defines and value and value2 for the values of those. This approach allows to adapt the kernels to different requirements at run-time. The queried kernel is saved in a map if it wasn’t queried before with the compile time defines. The map stores the name concatenated with the defines if those were passed to the function. This prevents that kernels are created multiple times with the same compile time arguments. If the kernel doesn't exist, the functions checks if a kernel with the name was loaded if this is not the case an error gets thrown. The pointers to every created kernel is stored in an array, if the kernel wasn’t created and is queried a null pointer is added to the array and the respective index is returned. The kernel is then stored in another map which stores all kernel objects that still need to be created, with its compile time arguments. The reason for saving a null pointer in the kernels array is that this way the index of the kernel can stay the same even if it is not created at this point. The kernel indices are then used in AddOperation which is used to add operations to the OpenCLBackend. The function can be used to either create a normal operation object or an increment operation object, depending on the template arguments. The function gets information about the number of work elements in each dimension passed to it and an argument which hints at how big a work group is. The work groups can also be changed to be a multiple of this parameter by the OpenCLBackend system, but no adjustment is made at the moment, since this would require a check, if the kernels executed correct, even if the work group size is a multiple of the given values. Therefore, the given value is used as work group size. There is also a function argument called OperationType that defines the direction of the operation. There are three different OperationTypes which are FORWARD, BACKWARD and UPDATE. They essentially define in which part of the deep learning system they are applied. FORWARD stands for the forward or inference operation of the computation graph, BACKWARD stands for the backward pass, the computation of the gradient of the computation graph, and UPDATE stands for operations that need to be done when all gradients where calculated, which are for example the application of the optimizer but may also be operations than perform clean-up of some Buffer objects or transfer data. The operations are stored in three different array objects, one for each

direction of computation. This way the OpenCLBackend does not need to know in which whey the operations are computed in a more explicit manner it only gets informed over the order through calls to add operations which adds it to the correct array. The result is calculated by iterating over the respective array, which is done in the Run function. The Run function is used to iterated over an array of BaseOpeartion pointer calling the Run function from each element of the array to Run the operation. Which pass is taken depends on the direction constant that is passed to the Run function. If profiling is enabled the OpenCLBackend has three arrays saving the duration of execution from every kernel that is called in each pass. Retrieving and storing of the times is then done in the run function of OpenCLBackend. The run function iterates over the forward and update array in forward direction and over the backward array from the end to the beginning, because all three operations are generated at the same time and the operations in the backward list are therefore executed in the wrong order. The operations are saved in an array, because arrays are quite fast to iterate over. Before run can be called it is necessary to call the PrepareRun member function of OpenCLBackend. This function is used to create all kernel objects, that are saved in needsToCreate. It separates the names of the kernels and the defines. Each Kernel is put in Source object and then the corresponding kernel is

```
void kernel ReLU(global read_only const float* restrict A, global write_only
float* restrict B, const int size)
{
#define GS 32 //One dimensional Work Group size

	const int i = get_global_id(0);
	const int tx = get_local_id(0);
	const int sizeX = get_local_size(0);
	const int groupId = get_group_id(0);

	if (i >= size)
		return;

	__local float buffer[GS];
	buffer[tx] = A[groupId *sizeX + tx];

	barrier(CLK_LOCAL_MEM_FENCE);

	B[groupId * sizeX + tx] = fmax(buffer[tx], 0);
}
```

Code 4.3 ReLU Kernel

compiled. The kernel objects are then retrieved and saved at the position in the kernel array with the corresponding index. An additional optimization would be to combine all not colliding kernels in one source object to reduce the compile time. The time spend on

compiling the kernel is a relatively small drawback compared to the ability to adapt the kernels and therefore being able to execute them in training faster. The OpenCLBackend also contains a few simple functions to write and read from kernel objects and to reset a kernel to a specific value via clEnqueueFillBuffer which is used to reset the gradient buffer objects after the update step.

4.1.3. OpenCL Kernel Collection

The OpenCL kernel collection contains kernel for many different basic operations which are used to calculate the outputs and the gradients of the NN on the GPU. Most operations have not only a kernel for a forward pass but also for a backward pass. The backward pass operations add the result always to the buffer. The reason is that one buffer may be the input for multiple operations, therefore the gradient is computed as a sum of multiple gradients. It is necessary to set every buffer, that is the result of a gradient computation, to zero. Every kernel is designed to compute the result for many different inputs at the same time, because in deep learning the training is performed on mini-batches. This has the benefit, that it allows multiple executions of the same algorithm in parallel, where the executions have no dependencies among each other. This is ideal for GPUs which are very good at highly parallel operations, allowing a better utilization of the available hardware. While most kernels in the collection are simple and don't deviate much from

```
void kernel MatrixMul(global read_only const float* restrict A, global
read_only const float* restrict B, global write_only float* restrict C, const
int hA, const int wB, const int wA)
{
	//Tile size definitions can be omited and set as compile time argument
at runtime

#define TILE_SIZE_X_2D 16
#define TILE_SIZE_Y_2D 16

#define WPTY 1
#define WPTX 1
#define TOTAL_WPT (WPTX * WPTY)

	const int tx = get_local_id(0);
	const int ty = get_local_id(1);
	const int lSizeX = get_local_size(0);
	const int lSizeY = get_local_size(1);

	const int gIdX = get_group_id(0);
	const int gIdY = get_group_id(1);

	const int i = get_global_id(0);
	const int j = get_global_id(1);

	float sum[TOTAL_WPT];

	for (int w = 0; w < TOTAL_WPT; ++w)
		sum[w] = 0;
```

```
        //initalize the local memory
        __local float localTileA[TILE_SIZE_X_2D * TILE_SIZE_Y_2D * WPTY];
        __local float localTileB[TILE_SIZE_X_2D * TILE_SIZE_Y_2D * WPTX];

        int numTiles = wA / lSizeX;

        for (int t = 0; t < numTiles; ++t)
        {
                //Load the tiles into memory.
                for (int x = 0; x < WPTY; ++x)
                {
                        int yIdx = gIdY * (WPTY*TILE_SIZE_Y_2D) + x*lSizeY + ty;
                        localTileA[tx + (ty + lSizeY * x) * lSizeX] = A[t*lSizeX
+ tx + (yIdx)* wA];
                }
                for (int x = 0; x < WPTX; ++x)
                {
                        localTileB[tx + lSizeX * x + ty * (lSizeX*WPTX)] = B[(t *
lSizeY + ty)*wB + (xIdx)];
                }

                barrier(CLK_LOCAL_MEM_FENCE);

                //Compute the different results
                //Compute the different results
                for (int k = 0; k < lSizeX; ++k)
                {
                        for (int wY = 0; wY < WPTY; ++wY)
                        {
                                for (int wX = 0; wX < WPTX; ++wX)
                                {
                                        sum[wX + wY * WPTX] += localTileA[k + (ty +
wY * lSizeY)*lSizeX] * localTileB[(tx + wX * lSizeX) + k*(lSizeX*WPTX)];
                                }
                        }
                }

barrier(CLK_LOCAL_MEM_FENCE);
        }

        for (int wY = 0; wY < WPTY; ++wY)
        {
                for (int wX = 0; wX < WPTX; ++wX)
                {
                        int xIdx = gIdX * (WPTX*TILE_SIZE_X_2D) + wX*lSizeX + tx;
                        int yIdx = gIdY * (WPTY*TILE_SIZE_Y_2D) + wY*lSizeY + ty;

                        //Write the results into the buffer
                        C[xIdx + (yIdx)* wB] = sum[wX + wY * WPTX];
                }
        }
}
```

Code 4.4 Matrix Mutliplication Kernel

each other, some of them are more complex and contain some special details. An example implementation of an element wise operation can be found in Code 4.3. This kernel code calculates the output of a ReLU activation function. Most activation functions and element wise additions, multiplications, etc. can be computed similarly. The specific computations, for the ReLU example the max operation, need to be replaced. The kernel

RUB

queries the local and global thread index, checks if the thread is in bounds of the buffer. Element wise operations are all linearized since the real dimensions are irrelevant for element wise operations. If the thread is in bounds, it loads
the parts of the buffer that are computed by the work-group into local memory performing coalesced memory reads. Then the actual computation is conducted, in this case ReLU and the result is saved in the output buffer. One thing to note is that all kernels make use of the restricted keyword [60], meaning that the pointer is essentially the only pointer,

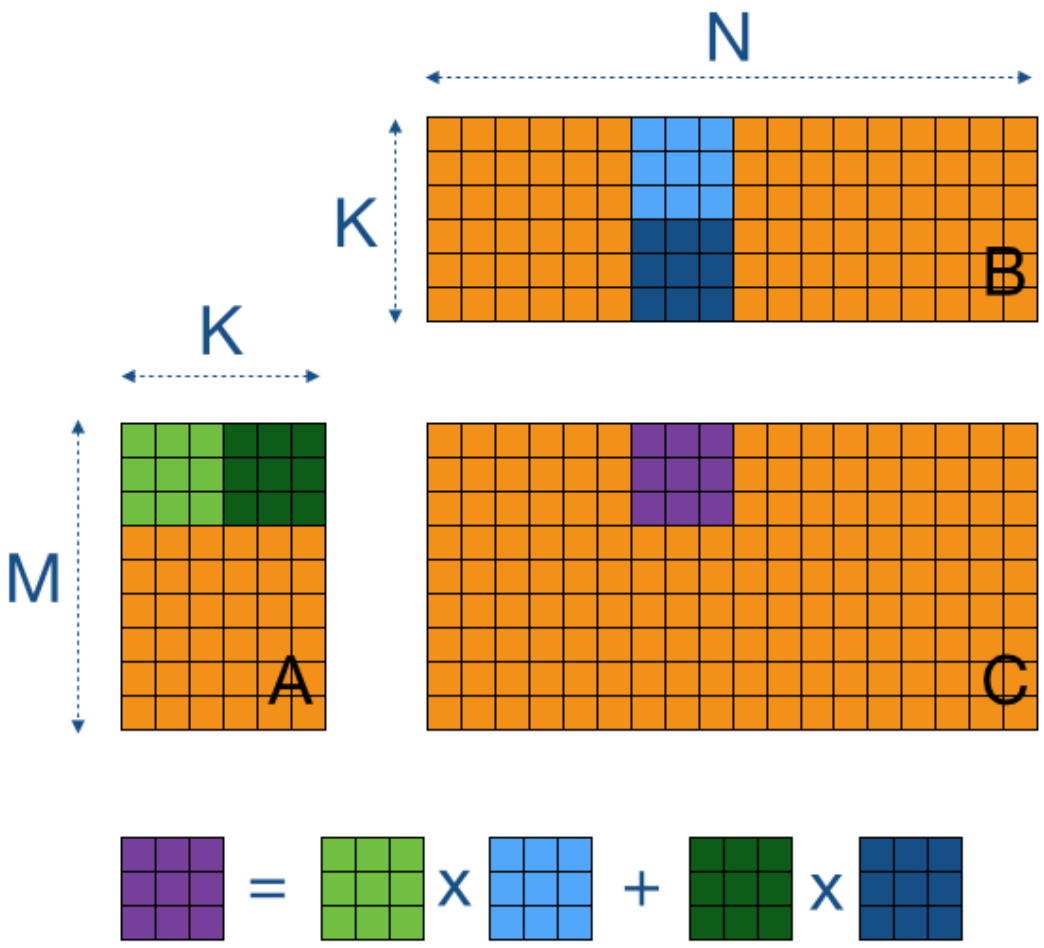

Figure 4.4 Tile Based Matrix Multiplication

that accesses its memory. This allows the compiler to optimize the code better, because it gets more information about how the variables and kernels are used. This restricts, in which how the kernels may be used by the system. For the use in NNs this is no problem because at training time, each result must be stored in a different buffer. At inference time, it is necessary to make use of double buffering to prevent any errors. Matrix multiplications and comparable operations, using multiple input elements to compute one output, require double buffering anyway. Fortunately, inputs to a kernel are most of the time different objects. The read_only and write_only keywords are used because, in

```
void kernel Convolution(global read_only const float* restrict A, const
global read_only float* restrict K, global write_only float* restrict C,
const int wA, const int hA, const int wK, const int hK, const int dK, const
int numK, const int pad, const int batchSize)
{
#define ROW_SIZE ((TILE_WIDTH-1)*STRIDE_X+WIDTH_KERNEL)

#define MEM_SIZE ((TILE_HEIGHT * TILE_WIDTH)/ROW_SIZE)

#define ROWS_PER_LOAD 15//(MEM_SIZE * WIDTH_KERNEL)

#define ROWS (ROWS_PER_LOAD / WIDTH_KERNEL)

#define TOTAL_KERNEL_SIZE (WIDTH_KERNEL * HEIGHT_KERNEL)
	const int imageSize = wA * hA;
	const int kernelImageSize = wK * hK;
	const int kernelVolume = kernelImageSize*dK;

	int imageOffset = imageSize;
	int batchOffset = imageSize * dK;

	const int tx = get_local_id(0);
	const int ty = get_local_id(1);

	const int i = get_global_id(0);
	const int j = get_global_id(1);
	const int k = get_global_id(2);

	const int lSizeX = get_local_size(0);
	const int lSizeY = get_local_size(1);

	const int groupIdX = get_group_id(0);
	const int groupIdY = get_group_id(1);

	//Allocation of local Memory
	__local float kern[TILE_WIDTH * TILE_HEIGHT];
	__local float imageTile[ROW_SIZE * ROWS];

	float sum = 0;

	//Size of the output
	const int outputXSize = ((wA - wK + 2 * pad) + STRIDE_X) / STRIDE_X;
	const int outputYSize = ((hA - hK + 2 * pad) + STRIDE_Y) / STRIDE_Y;

	const int imgRemainder = (wA + pad + (STRIDE_X*lSizeX) -
	1)/(lSizeX*STRIDE_X);

	//Calculation of the start poisition in Image space
	const int startX = (groupIdX%imgRemainder) * (lSizeX*STRIDE_X) - pad;
	const int startY = (groupIdX / imgRemainder)*STRIDE_X - pad;

	const int unrolledPos = tx + ty * lSizeX;

	//Calculation of the read position
	const int posX = ((unrolledPos)) % (ROW_SIZE);
	const int posY = ((unrolledPos)-posX) / (ROW_SIZE);

	int imgRow;
	int img;
	int imgCol = startX + posX;
	int tmp;

	for (int l = 0; l <kernelVolume; l += ROWS_PER_LOAD)

	{
		kern[unrolledPos] = K[j * kernelVolume + l + tx];
		tmp = posY + l / WIDTH_KERNEL;
```

RUB

```
                img = (tmp) / HEIGHT_KERNEL;
                imgRow = startY + tmp - img * HEIGHT_KERNEL;

                imageTile[unrolledPos] = A[imgCol + img * imageOffset + imgRow *
wA + batchOffset * k];

                barrier(CLK_LOCAL_MEM_FENCE);
                for (int m = 0; m < ROWS; ++m)
                {
#pragma unroll WIDTH_KERNEL
                        for (int n = 0; n < WIDTH_KERNEL; ++n)
                        {
                                sum += imageTile[tx * STRIDE_X + n + m * ROW_SIZE]
* kern[m * WIDTH_KERNEL + n + ty * lSizeX];
                        }
                }
        }

        C[tx + (startX + pad) / STRIDE_Y + ((startY + pad) / STRIDE_Y +
outputYSize * (j + numK * k))*outputXSize] = sum;
}
```

Code 4.5 Convolution Kernel

general, it is only read or written to the memory pointed at by the pointer arguments, an exception being the result of gradient operations. This again allows the compiler to optimize the code and therefore possibly gaining a reduced training and inference time.

The next kernel that is worth looking at in more depth is the kernel for matrix multiplication. It is often used in NNs especially, when RNNs are implemented. LSTMs and GRUs make heavy use of them. Convolutions can also be reduced to a matrix multiplication, but in the current implementation, the convolution kernel implements a custom matrix multiplication. Code for a matrix multiplication can be found in Code 4.4 and Figure 4.4 visualizes how the algorithm works making use of a tile based approach. The implementation of the matrix multiplication was inspired by [79]. To make the general workings of the implementation clear, boundary conditions very excluded in the displayed implementation and it is assumed that the sizes are a multiple of the work-group size. This is not necessarily a restriction on the algorithm because the input matrices could be padded with zeros to coincide with the work-group size. This would allow the direct use of it, in the way it is displayed. The algorithm splits the output of the matrix multiplication into two dimensional tiles where each dimension is a multiple of the work-group size big. The factor is determined by how much work each thread should do. It allows each thread to not only calculate one output pixel, but 2 or more in each dimension. The work-size adds an additional parameter that can be adapted to possibly increase the throughput of the algorithm. Allowing each thread to compute multiple results increases the ratio of loads to performed instructions, because multiple data points can be loaded

into registers at the same time and can then be used to compute multiple results. The tile size and the work per thread must be defined at the compile time of the kernel. The algorithm then creates a private array of floats in which the intermediate results are stored, the size is the total work each thread should do. Then two local memory objects are created one is used to load a tile of matrix A into the local memory and the second is used to load a tile of matrix B into local memory. The size of the allocated local memory is an equal to the tile size times the amount of work per thread.

The algorithm splits all columns of A into tiles, with the size of the tile size in x direction and all rows of b also in tiles, the height of the tiles also determined by the tile size in x direction. The algorithm then loops over the tiles in A and B. Each iteration, the threads of the local work-group load one complete tile of A and B into local memory. When the work per thread in x or y dimension is bigger than one, each thread must load more than one element of the total tile into memory. This is done in a fashion that supports coalesced memory accesses. This maximizes the used memory bandwidth and minimizes the number of global memory transactions, that are needed. To achieve the desired memory access, the total tile is split up into work per thread tile parts in one dimension. All threads of the work-group load one complete part of the tile in an aligned manner, followed by the next and so on. Since the start of each tile and sub part of a tile, is always a multiple of the work-group size, in the respective dimension and this size is chosen to be a multiple of the warp size, this archives coalesced memory access. After both tiles are completely loaded into local memory the multiplications and additions to calculate the results of the matrix multiplication is conducted.

Then a new iterator of the loop begins and the next tiles are loaded into local memory. This is done until all tiles, in x direction for A and in y direction for B, where multiplied. This computes the result of the matrix multiplication for the tile that is calculated by the work-group. The result is then stored in the respective output locations in global memory in C. To incorporate boundary conditions an adaption to the loop and the loading into local memory, is made. The loop is split up into two parts. The first part is the same loop as before, but only calculates the results for all tiles that fully fit into local memory, in the x direction of A. The second part then loads the remaining tile in local memory and calculates the final result. This way the branching can be reduced on the cost of prolonging the length of the code. The adaption made to loads from global memory into local memory is, that if a load would access memory that is out of bounds of the respective matrix, a zero is written into the local memory. In this way, the number of branches to is restricted to only one position.

The last kernel that is discussed in detail is the convolution kernel. It also makes use of tiling, but is in comparison much more complicated than the matrix multiplication kernel, due to the transformation that needs to be performed. It is necessary to create two different types of kernels, because in the forward pass, the filter maps are in most cases fairly small. Typical sizes are $3\times3, 5\times5, 7\times7$. In the backward pass, convolutions with big filters must be performed. To compute the gradient with respect to the filter, a convolution of an image, with an image sized filter must be performed. The image sized filter is the gradient. Those two different situations need a different approach and different designed kernels to archive a good utilization of the available resources. In order to understand the design of the kernel for small filters one has to take a look at the unrolling transformation. The filters can be saved in this transformed way, since this is a natural way to store the different filters. For the input images is this not the case, they need to be transformed in place in the local memory. The global memory requirements can be decreased. The question is, how this can be done in an efficient way. One observation that can be made is the way, in which the data is duplicated, through the transformation. Taking a look at Figure 2.13 it can be see that the filter width number of rows, in the transformed input, have the same content in one row but with an offset of one. A stride bigger than one let the different rows contain the pixels that are skipped in the other rows. The last statement assumed that there are no pixels that are never used by the convolution, which is true if the stride is smaller than then the width of the filter. For a filter width of three and stride one this means, that three rows contain the same pixels but with an offset of one. The second observation is that pixels in one row are the pixels that are either adjacent to each other or the size of the stride separated from each other. Assuming a filter with a stride of two and a filter width of three. The first four elements in the first row are separated by one element. The second row contains the elements that where skipped in the first row while the third row contains the same elements as the first row but with an offset of two. This observation lead to the following design of the kernel and the restrictions made on it. A basic code implementation that ignores boundary conditions can be seen in Code 4.5. The kernel splits the output into multiple tiles where each tile is calculated by one work-group. The kernel splits the unrolled columns of the filter into tiles which have a width, that is a multiple of the filter width. The height is equal to the work-group size in y direction. The kernel loops over all tiles in x direction that are created this way. The input image is also split into tiles where the x direction has the size of the work-group in x direction. The height of the tile is equal to the width of the filter tiles. The kernel loads at every iteration row parts from the position at the offset, of the input, into local memory.

Each row part in local memory is used to represent the width of the kernel number of rows of the transformed matrix.

This way adjacent pixels of the input are read together even though they are used at multiple different parts of the input. This archives data reads in a coalesced way even though multiple memory transaction may be necessary. Multiple rows are loaded at one step. Therefore, the row size is bigger as the respective tile size. This approach of loading the kernel and input image into local memory leads to a few restrictions and requirements on how the kernel should be used. The first requirement of the kernel is, that the width of the output should not be too small, for it to be efficient. Ideally it should be a multiple of the work-group size. The second is that the width of the kernel should not be too big while the height of the kernel can be more flexible. The kernel works in the following way:

It creates the local memory for the image and the kernel tile. Then it loops over the tiles in the kernel that have the same y value and over the tiles in the image that have the same y value. At each iteration, the data of the tile is loaded into local memory as was described before. Then each row of the kernel's local memory is multiplied with one column of the image's local memory, but adhering to specific addressing rules for the image, by one thread. The result is always added to the private sum variable that is initialized with zero and used to calculate the result of one thread. The addressing is different because the filter width number of rows must be loaded from the same row into local image memory but at an offset depending on the stride and which row, in the image, this represents. This way the rows that where loaded from the global memory into local memory can be used multiple times. In the next loop iteration, the next tile is loaded into local memory and is used the same way until total result is reached and saved into the global output array C. This kernel again uses the same separation of the loop as already did the matrix multiplication, to adhere to boundary conditions. Out of bound reads are also handled the same way by saving a zero into the memory. One major thing to consider is that a row that is loaded into local memory can only contain one row of the input even though the created matrix would continue and just load the input from the next row. This introduces additional boundary conditions and is the reason for the first restriction regarding the number of output pixels. Threads that would calculate the result which falls into the next row just stay idle and therefore reduce the overall utilization of the resources. If the number of output pixels is big, the influence is relatively low, this suggests that a small work-group width could proof to be useful. The total work group size should stay to be a multiple of the warp size.

The computation of the gradient with respect to the filter, behaves different because, the filter is only two dimensional, but it must be applied to many different images, e.g. every image in the batch. The results, of those convolutions, must then be summed together. This way the number of filters that must be unrolled is only one but the filters are very big. They can't be completely loaded into the memory. The filter matrix must be split into tiles that don't align with the unrolled kernel positions as was the case before. In this case the input matrix must be transposed, because the gradient with the respect to the original filter is calculated. Since the matrices are created in local memory it is not possible to transpose them directly. It is necessary to create a kernel which takes this into account and directly creates the transposed matrix. The observations that were made before also need to be adapted to this new situation. In this situation, the number of columns of the image matrix is equivalent to the product of the width and height, of the original filter. It is therefore often rather small. The number of rows is equivalent to the product of the output width, height and the batch size, which is rather big and reduces the possible parallelization. One benefit is that the number of different filters is equivalent to the total number of output feature maps, hence increasing the possible parallelization quite much. It is again worth to take a closer look at the construction of the matrix to find a way how to increase the memory efficiency. The situation is relatively like the one before, but now the same memory addresses are read into the columns and not into the rows. They are again at a specific offset and but this time there is not stride, so it stays the same. The stride is not implemented for the gradient convolution, with respect to the weight parameters. The reason is that the stride requires, loosely speaking, holes in the input because not all input pixels influence all weight elements. This could be solved by padding the input, but this is not implemented at the moment. A convolution with stride different from one would not solve the problem. One difference is, that the total number of rows that must be loaded could have been flexible, but it needed to be a multiple of the filter width. Now it must always be equal to the work-group size in x dimension. This introduces additional problems, which have to be taken care of, because adjacent work-groups must respect the initial start. The initial start position might be unaligned with the columns, that are equal to each other, therefore a duplication in different work-groups is necessary. The resulting offset must also be accounted for. The adjacent data is read into rows and not into columns because this way the data can again be read in a coalesced manner. The access is comparable to the first convolution kernel with equal properties regarding coalesced reads. The kernel implementation is very similar to the first convolution, but without the stride.

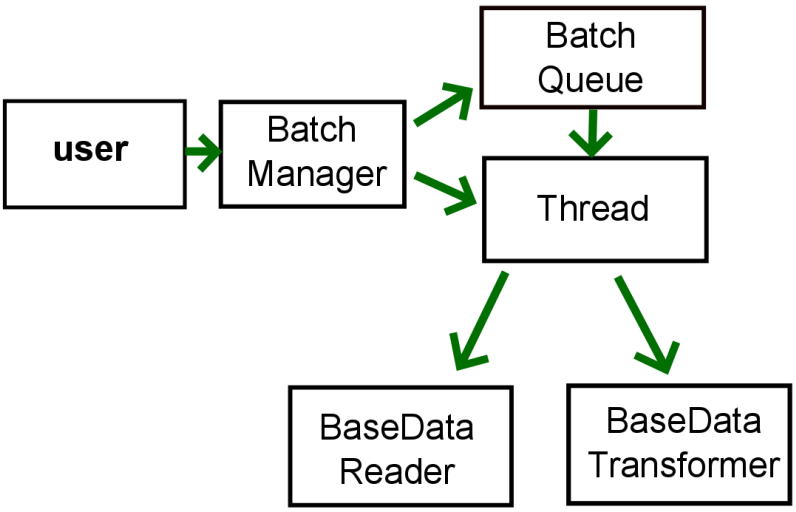

Figure 4.5 Data System Architecture

4.1.4. Data Sub-System

Deep learning needs a big amount of data, which makes the handling of the data an important topic itself. While some small data-sets like MNIST [27] fully fit into the main memory. Other data sets, for example the IAM [80] data-set might not fit into memory. They need to be read into memory from the hard drive dynamically. Additional to loading the data, it is also necessary to pre-process it and adapt it to the given situation. Loading data from the hard drive and pre-processing it can be a very slow process. It could possibly increase the training time, if not done correctly, e.g. on the same thread which also controls the training. A solution to this problem is to manage the reading and pre-processing of data on one or multiple threads asynchronous. The computation graph runs, in most cases, on the GPU and one thread, the main thread, is used to manage the execution of it. Other threads could therefore be used to read the data. Modern processors contain often more processing cores than one so that additional threads can run on some of those cores. The main thread, which manages the execution of the OpenCL kernels can then run on another core. To allow easy use of this asynchronous data reading the Data Sub-System was created. It consists of four components (Figure 4.5), the BatchManager, the BatchQueue, the BaseDataReader and the BaseDataTransformer. The BatchManager class is used to interface the Data sub-system from the rest of the system, it creates the threads, and controls how data is loaded. The BatchQueue class controls where the data is stored and where it is loaded from, while it also synchronizes data access. The BaseDataReader is an interface for custom data readers and the BaseDataTransformer is an interface for custom data transformers. The BaseDataReader

is used to load data directly from files and so on, while the BaseDataTransformer is used to pre-process the data in a specific way. The last two are more general because every data-set is differently stored and needs to be pre-processed in a different way. The system makes also use of Tuples because they allow the data loader to return arbitrary types of data and at the same time also an arbitrary number of elements. This is useful because non-sequential data often only needs two types of returns, which are the data of the batch and the label. This changes in the sequential setting, for various length sequences, as in the case of handwritten character recognition. In this application, the returned values for a batch need to contain, beside the image and the strings of labels, the length of the sequence for every example. Another example would be multiple different inputs that are inserted at different points in the computation graph. The Tuples are used in a different way than before, because in the backend they were used directly with the variadic template arguments to represent the data. There was only the need to store single arguments for the kernel functions, but now the data is gathered in batches. Each element of the variadic template must be stored once for each batch element. There are two ways how the data for the batches can be stored. To make the explanation easier, it is assumed that the data consists only of images as data and corresponding class labels. The first approach would be to store an array of Tuples in each batch. Each tuple then saves exactly one example of image data and one corresponding label. This is problematic, because the data that is transferred onto the GPU must be sequential. To archive this, the separated data entries saved in the Tuples must be merged, if this approach is taken. The same holds for the labels. This would incur an additional overhead because an additional pre-processing step is necessary, especially if the number of elements in the tuple grows. This additional pre-processing reduces the effectiveness of the pre-processing on different threads. The alternative approach would be to create a tuple out of arrays, which contain the data and the labels. This would prevent the problem that occurs with the first approach. Practically this approach makes it not necessary, that the user specifies the variadic tuple as array but only the data types of the data and label. The variadic template is then converted into a Tuple of arrays of the variadic template arguments.

All parts of the system are templatized with the variadic template, but at all places Tuples of arrays are used. This approach abstracts away some of the details of the underlying implementation and would allow the usage of different container types. The user always gets a pointer onto a Batch object returned, which consists of a tuple of arrays and some extra information related to this batch. Even though this approach is very flexible one thing must be kept in mind. Each entry in an array, that is returned must have the same

size. This is in some sense also necessary for the GPU, because the kernel implementations on the GPU expects all elements, of the input to be off equal size. This restriction is therefore no big problem, but must still be accounted for through padding.

The most basic parts of the system are the BaseDataReader interface and the BaseTransformer interface. The BaseDataReader is a basic interface for deriving classes that are used to load data from disk or from other arbitrary sources. It consists of three abstract functions that must be implemented by derived classes. Those are AllocateCopy, GetNextData and AddOffset. AllocateCopy is used to create copies of the class in a specific way, which is chosen by the programmer. It is necessary to create an instance for each thread. The AddOffset function changes the current data point that is retuned from the data-set. The implementation depends specifically on the data format and how it is loaded because some data is read from disk while other may already reside completely in memory. The last function GetNextData (Code 4.6), is the function that should return all elements of one batch. Passed to it are a Tuple of arrays of the variadic template, an array of arrays of an integer type and an array of arrays of SizeVec. The data that is read in the data reader does not necessarily need to have any alignment and every example might have a different size. In the case of a dataset of images of handwritten words each image might have a different size and each label sequence has a different length because the number of characters in each word differ. To account for this fact the second and third function arguments where introduced. While the first argument is only used to output the data arrays, the second parameter is used to represent at what offset the data example is positioned in the respective array. The third returns the size of it. Each contains the number of variadic template arguments, number of arrays. This interface is implemented for two different data sets, the MNIST data-set and the IAM data-set, which are both good examples for how the functions can be implemented and used with this approach. The MNIST data-set fits fully into memory, while the IAM data-set is too big to be loaded completely into memory and is therefore read from multiple files on the fly. One thing to note though is that the images in the IAM data-set consist of many files which results in some overhead because the operating system needs to be used for every example to be loaded. An alternative would be to load all examples in a pre-processing step and then write them into one file, eventually without compression, or to write them to a few files,

```
GetNextData(BackendSystem::Tuple<std::vector<varT>...>&                    data,
std::vector<std::vector<size_t>>&                                        offsets,
std::vector<std::vector<NNSystem::SizeVec>>& sizes);
```

Code 4.6 GetNextData Function

making use of some specially sorting. The different files can then also be used as offset so that different threads start with different files.

The BaseDataTransformer is used to transform the loaded data into specific representations. Example transformations are padding all images of a batch to the same length or to use it for data augmentation. It contains two different abstract functions which are Transform and AllocateCopy. AllocateCopy serves the same purpose as it did before

```
void Transform(BackendSystem::Tuple<std::vector<varT>...>& dataOutput,
std::vector<NNSystem::SizeVec>&, BackendSystem::Tuple<std::vector<varT>...>&
data, std::vector<std::vector<size_t>>& offsets,
std::vector<std::vector<NNSystem::SizeVec>>& sizes);
```

Code 4.7 Transform Function

in BaseDataReader. The Transform function is the function, that transforms the batch into an appropriate representation (Code 4.7).

It specifies 5 different function arguments which are the output data, an array of SizeVec and the same arguments as GetNextData. The last three arguments are in principle the output of the GetNextData function of BaseDataTransformer and are the input for the transformation. The first argument is the output tuple that was create by the transformation and the second argument contains the four-dimensional size of the elements in each array in the Tuple. Both tuples that are passed to the transformation function are not allowed to be the same object. There exist two derived classes of this abstract class, one for the MNIST data-set and one for the IAM data-set. The implementation for the MNIST data-set does not do much in the transformation function. The IAM data-set function is used to pad the input images and labels into a specific size. The Data sub-system should ideally allow the use of multiple threads to prevent, that the data reading and pre-processing are the bottleneck of the learning pipeline. It is beneficial to always store a specific number of batches so, that it is always possible to retrieve a batch when it is needed. The creation, storage and loading of this data needs to be synchronized if multiple threads are involved. The class that provides the system with these abilities is the BatchQueue. The BatchQueue contains multiple functions to store and retrieve data in a thread safe way using mutexes. The system uses a memory pool of batches. For this purpose, it contains three different data collections which are used to store the batch data, the memory pool, a list of batch elements, that must be new created and a list of fully created batches. The batches are stored as object in a basic struct, called “Batch”. The object stores the tuple of arrays, the size of the batch and an array of SizeVec for the size of an element in each array in the tuple. The w component in each SizeVec

object stores the length of the sequence if the data is sequential. All batch objects that are used in the framework are stored in this class. All batches that are retrieved are pointer onto elements in this array. The reason for this approach is, that even though the array objects, that are stored in the batches might change, the space that is allocated by those array does not necessarily shrink, even when the array size shrinks. This reduces the need for reallocations only to situations where it needs to grow. When a batch is retrieved from the BatchQueue, an index is always returned additionally. This index is used to "return" the object back to the BatchQueue. While this might increase the risk of wrong usage by the user, is it not so big of a problem, because the user does not have direct access to the BatchQueue and the indices. Only the pointer is returned by the system. The user can ignore it when it has served its purpose. All the returning of batches and saving of the indices is done in the BatchManager. Only one implementation that uses this approach correctly is needed. It would also be possible to expand this system by allowing the pointer to be returned directly. This could for example be implemented through a dictionary that maps all pointers to the correct indices. The indices must be retrieved because they are saved in the two collections that signal if a batch must be created new or is ready to be used. signalling something different to the BatchQueue. If a batch was used and is not needed any more, the index is returned to the BatchQueue signalling that it is not needed any more. The index of the object is then pushed onto the queue. Later a thread will retrieve a batch that is not needed any more and fill it with a new batch of the data-set. To synchronize accesses to the two collections mutexes, for each one, are used to lock the collections when they are accessed. The functions GetBatch and GetEmptyBatch retrieve batches from a corresponding collection. GetBatch returns a batch that is in ready to be used and GetEmptyBatch returns a batch that needs to be created. Both functions wait until the respective collection contains an element using sleep. The time they sleep is different, because the batch loading threads have only the task to create batches, while the main thread runs the neural network and manages it. It therefore is not allowed to sleep as long since shit would stop the training completely. The time the thread sleeps in GetBatch is therefore shorter by a magnitude. There are two functions to return batches which are ReturnBatch and ReturnEmptyBatch both access again the corresponding collections. Those functions only lock the respective mutex and then add the index to the collection.

```
BackendSystem::Tuple<std::vector<varT>...> resultTuple;
                    std::vector<std::vector<size_t>> offsets;
                    std::vector<std::vector<NNSystem::SizeVec>> sizes;
                    for (size_t i = 0; i < numArgs; ++i)
                    {
                           offsets.push_back(std::vector<size_t>());
                           sizes.push_back(std::vector<NNSystem::SizeVec>());
                    }

                    BaseDataReader<varT...>* reader = baseReader-
>AllocateCopy(); //Every thread needs its own custom copy of this class
because the loader might use files etc. also class members are changed
                    BaseDataTransformer<varT...>* transformer =
baseTransformer->AllocateCopy();//Every thread needs its one custom copy of
the class because class members are changed and they should be independent of
any threading
                    reader->AddOffset(threadIdx * 1000);// All threads should
start at different positions in the data

                    size_t idx;
                    while (!finished)
                    {
                           Batch<varT...>* batch = queue.GetEmptyBatch(idx);
                           batch->sizes.clear();
                           TupleLoop<0, numArgs-1 , function,
std::vector<varT>...>::apply(resultTuple);
                           TupleLoop<0, numArgs - 1, function,
std::vector<varT>...>::apply(batch->data);


                           reader->GetNextData(resultTuple, offsets, sizes);
                           transformer->Transform(batch->data, batch->sizes,
resultTuple, offsets, sizes);

                           for (size_t i = 0; i < numArgs; ++i)
                           {
                                  offsets[i].clear(); //Does not necessarily
reduce size (Depends on Compiler-> Custom Class)
                                  sizes[i].clear();
                           }
                           queue.AddBatch(idx);
                    }
```

Code 4.8 ThreadRun Function

The last component, the BatchManager, makes use of all the other components and controls especially how the indices from the BatchQueue are handled. It also creates the threads and provides the function, which they run. It has one constructor to which, besides the configurations of the queue, a pointer on a BaseDataReader and on a BaseDataTransformer is passed. The arguments are saved in different members and the BatchManager creates a BatchQueue. It is the only class that has as additional template argument the number of elements in the variadic template. This is necessary to perform operations on all elements of the Tuple of arrays. The constructor then starts the number

of threads that where passed to it. All threads run the ThreadRun function. The argument to it is the index of the thread which is later used as offset, Code 4.8 displays the basic working of this class.

The function begins by creating a temporary Tuple which is used to store data that needs to be transformed. It also creates the needed array of arrays for the offsets and sizes. It creates copies of the original reader and transformer pointer via the respective AllocateCopy functions. This is necessary because each thread needs its own copy of the objects to work with different offsets and to have each allocate its one resources. Otherwise additional synchronization would be necessary, if for example the same file is accesses. This would slow down the whole data retrieval process. It also applies an offset on the reader via AddOffset. After all of this is done the thread starts its main loop which it runs until an atomic Boolean called finished is set to true by the main thread. The function retrieves a batch from the queue, stores its index and performs some clean-up operations on the temporary tuple and the retrieved batch. If the batch was used once before then the data in it is already set, which is not whished for. To clean up Tuple objects a loop class calls the clear function on every element in the Tuple. The loop class is also the reason why the number of template parameters was necessary. After this is done the thread retrieves new data from the BaseDataReader passing it the temporary Tuple. The data that is read into the tuple is then transformed via the BaseDataTransformer object passing it the temporary Tuple as input and the Tuple of the retrieved Batch as output. The batch is now initialized and can be used. It is returned to the BatchQueue using the index that was returned when the batch was retrieved. After the loop is done, the thread deletes its versions of the BaseDataReader and BaseDataTransformer. It then shuts down. Another important function that this class provides is the GetBatch function which returns a pointer to a Batch object. It is used to retrieve a batch from the queue and at the same time returning the last retrieved batch. For this reason, it stores the index of the last retrieved batch in a member object. Whenever a new batch object is requested from the BatchManager the old one becomes invalid. This way the user does not need to attend to the index of the batch, making the code used for training easier and more direct.

4.2. The Handwritten Character Recognition Neural Networks

Deep Learning offers a wide variety of different algorithms and layers which can be used to create different Neuronal Networks. What types of layers and algorithms should be used strongly depends on the input and output data, CNNs for example excel at tasks that are related to images while RNNs excel at tasks related to sequences. The task of

Handwritten Character Recognition incorporates both images and sequences. The input to an NN, that should perform Handwritten Character Recognition, is an image of handwritten text and the output is a sequence, namely the sequence of characters which are written in the image. For this reason, all NNs used in this work consists of two parts. The first part consists of convolutional layers, that perform transformations on the input image and create an encoding of the input. To generate a sequence of characters the second part of the NN is made up of recurrent layers. Three different NNs architectures were developed to perform handwritten character recognition. All three of them consist of the two basic parts but the way they are connected to each other, the loss functions and how the input image is passed to it differ. The first most basic model is used as baseline for the two more advanced systems. The first basic NN uses the encoding created by the CNN as initialization of the state. A sliding-window based NN is the second approach and is than expanded upon, by adding an attention mechanism, in the third model. The configurations for the conducted experiments will be explained at the end.

4.2.1. General Neural Network Implementation

While the three architectures differ in some points quite much, there exists some common ground between all three of them beside the fact that all of them consists of a CNN part and a RNN part. All CNNs that are used have a similar structure deviating only in the number of feature maps that are used. Each CNN consists of a complex structure that is repeated three times (Figure 4.6). Each complex structure consists of four operations which are a convolution, an addition of the bias to every feature map, an application of the ReLU activation function and a max-pooling operation. The convolution is performed with a filter size of 5×5 because the CNN consists of only three layers. Using a filter of size 3×3 would result in a relatively small receptive field so this way the total receptive field is bigger, giving the CNN the chance to extract bigger high-level features. More layers are not possible because of hardware limitations for the current implementation. The convolution has a stride of one and uses two padding bytes on all sides, so that the output images have the same width and height as the input images. As activation function is the ReLU function used because it supports a better flow of the gradient compared other activation functions like the Sigmoid activation function. It has proven to be very effective in practice, as was stated before. The max-pooling operation is performed with a window size and a stride of two in each direction. The output of the complete CNN part is flattened into a one-dimensional vector, for each batch element. This results in a matrix which

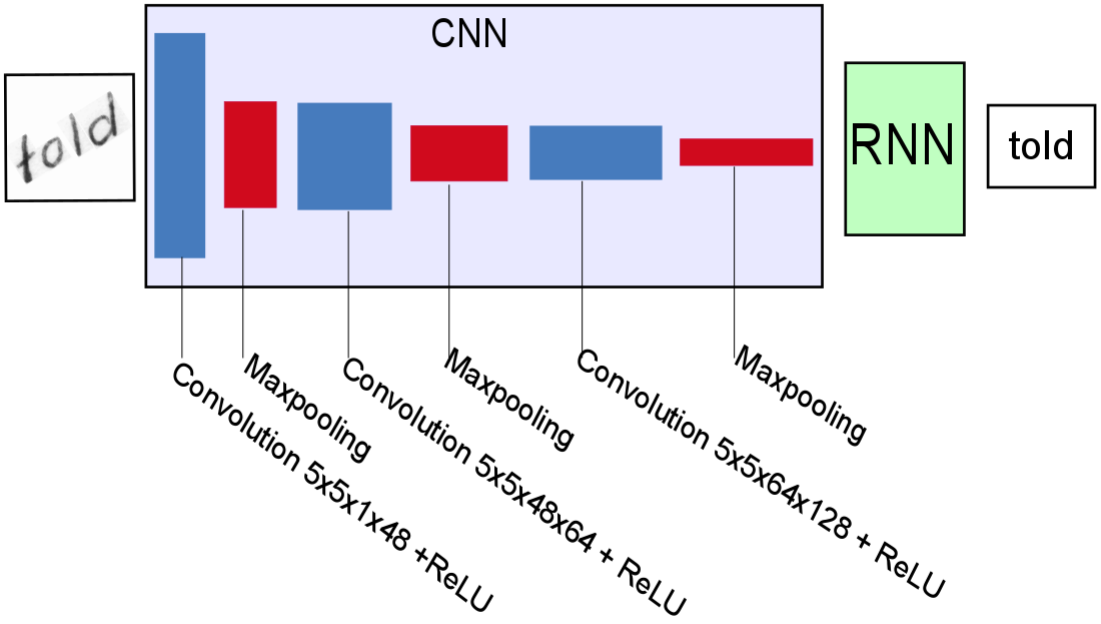

Figure 4.6 General Neural Network Architecture [80]

allows the application of matrix multiplication to it. Each RNN layer uses GRU cells as recurrent cells because they have proven to be useful in practice and are cheaper to compute. Each RNN contains one or multiple layers of GRU cells, where the output of one RNN layer is the input to the next.

All characters that are used as input or output to the NNs are represented as One-Hot encoding. The encoding makes an order of the characters which are part of the alphabet necessary. The One-Hot vector representing a character consist only of zeros, except a one at the position of the number of the character. The vector of the first character only consist of zeros and a one at position zero while, the vector of the second character only consist of zeros except at position one. An alternative representation would be to only input a single integer which has the value of the position of the character, but this approach does not work so well. It implies some form of similarity between characters close to each other, while characters being at very far apart position are less similar. The One-Hot encoding on the other hand does not imply such similarity between numbers in the encoding therefore allowing the NN to learn a metric of similarity. The output of the last recurrent layer is used as input to a fully connected layer which reduces the dimension to the total number of characters which are possible in the output. The result is then feed

into a Softmax function which computes the probability of each character. The complete architecture can be seen in Figure 4.6.

All three NNs are trained with mini-batches, but with different mini batch sizes. The reason for this is that bigger batches need more device memory, which is not available for the more complex NNs. The more complex NNs need more memory for each batch element than a simpler NN. Optimization of the NNs is performed using the Adam optimizer, for it making use of momentum and an adaptive learning rate, which makes a custom learning rate schedule unnecessary. To calculate the gradient BPTT is used.

The IAM dataset [80] is used to train and compare the models against each other. The data set is made up of images of English handwritten characters and the respective labels.

Figure 4.7 IAM Dataset Example [80]

There are different versions of this data set each having a different complexity. In the most basic version, used in this work, the images contain only complete words, whereas more complex versions contain complete sentences and paragraphs. The reason that only the most basic data set is used is that more complex tasks need more computational resources and especially much more time and memory which was not available on this scale. An example of an element in the data set can be seen in Figure 4.7. The data set contained 78 different characters in total. This is also the output alphabet of the NNs plus an additional *end* token which signals the end of the word. At the end of each label sequence an *end* token was added. In the data set are examples which were not correctly segmented. Those examples were not used in any way. The data set consists of 96454 examples in total which were split into three different parts. The parts are a training set, a validation set and a test set with the following number of elements 2472, 1426 and 92556 respectively. The training set is used to train the NNs, the validation set is used to decide when the training should end and the test is used to compare performance between all the NNs and the results of related work.

Each sequence of one hot character encoding in a batch must be of the same length because the RNN cells can only run complete batches. This is archived by padding each sequence in a batch with a one hot encoding of zeros until all sequences have the same length as the longest sequence in the batch. The Character Error Rate (CER) is computed by calculating the total number of permutations necessary to transform the returned word into the correct word and dividing this by the length of the correct word [53]. Permutations are substitutions, deletions and insertions. It is important to note that the CER can get higher than 1.

4.2.2. Encoder Decoder Neural Network

The Encoder Decoder Neural Network uses the CNN part to transform the input into a useful representation, which is then used to initialize the state of the first RNN layer (Figure 4.8). The output of the CNN is multiplied with a parameter matrix that has as output size the number of RNN cells in the first layer. The output of the NN, at each time step, is the probability of every character in the output alphabet. The number of steps is equal to the number of characters in the input image plus the additional *end* token, at training time. At inference time, the RNN runs until it outputs an *end* token which signals that the NN has outputted all characters it recognized in the input image. The input to the RNN, when the NN is used for inference, is the character that had the highest probability at the last time step. When the model is trained, the input is the correct character from the last time step independent of the output at the last time step. The loss function used to train the NN is the cross entropy because there is a correct character for each time step and the probability of this character should be maximized. The output of the CNN is multiplied by a parameter matrix to initialize the RNN, therefore each input image must

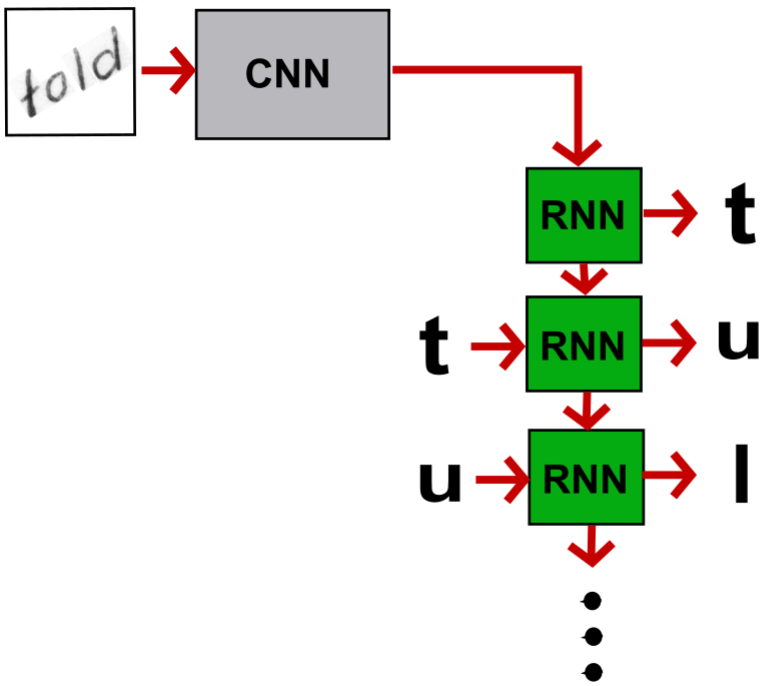

Figure 4.8 Encoder Decoder Architecture [80]

have the same size. This way the output of the CNN has the same size for every image. This is archived by padding all images of the IAM data set with zeros, until they have the

same size as the maximal sized image of the data set. In order to increase the speed of the training, the training examples are sorted into four non-overlapping buckets. Each bucket has a minimal and maximal sequence length and each example is put into the bucket, in which its sequence length fits. This technique is called bucketing [81] and allows the reduction of padding and especially unnecessary RNN steps. All elements in a batch are taken from the same bucket and have therefore more similar lengths.

4.2.3. Sliding Window Neural Network

The Sliding Window Neural Network works quite differently than the Encoding NN because the input to both parts, the CNN and RNN, is different. The input image is split up into multiple, possibly overlapping, windows along the width of the image. This is inspired by an approach to speech recognition in [43], where a speech signal is split up into multiple windows. The height of the window is always equal to the height of the original image but the width of the window and the stride are hyperparameters that must be decided upon. Each window is passed as input to the CNN and the resulting output is used as input to the RNN (Figure 4.9). If the windows overlap some work in the CNN is repeated because the CNN is applied more than once on the same pixels. Creating the windows using the output of the CNN therefore decreases repeated work and possibly the

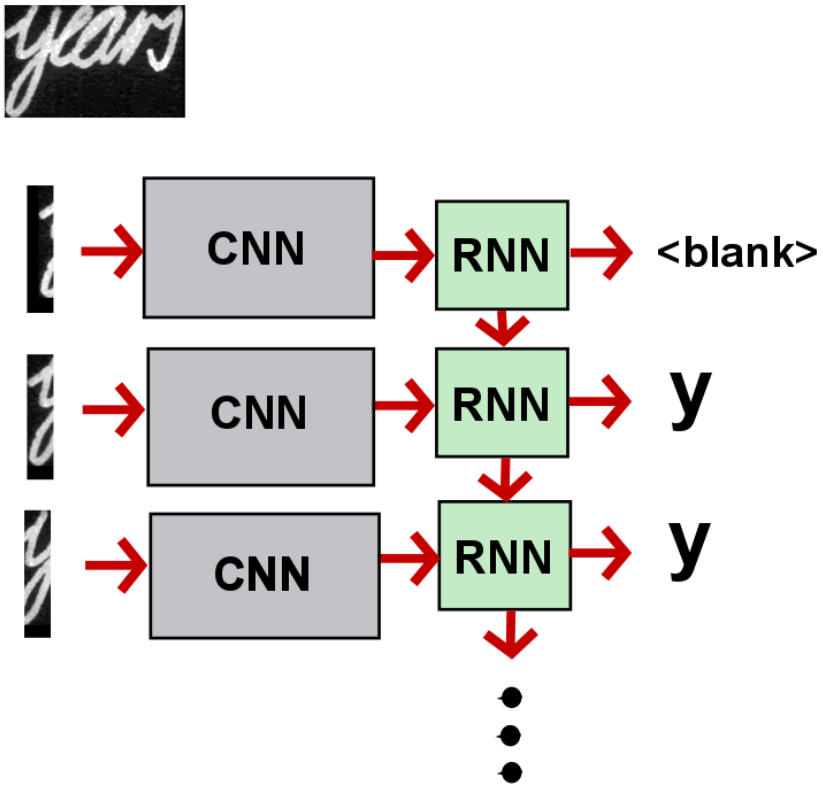

Figure 4.9 Sliding Window Architecture [80]

time training or inference takes. The used framework did not support the generation of windows in this way, therefore the windows were created using the NN input. The number of windows determines the number of time steps for which the RNN runs. This is the main difference to the Encoder NN because the number of time steps in the Encoder NN is only determined by the length of its output sequence and independent of the size of the image, in contrast the number of time steps for the Sliding Window NN only depend on the size of the input image. The state of each GRU cell layer is initialized with zero. Every time step of the Sliding Window NN outputs only one character. The number of windows must at least be as big as the output sequence, otherwise it is not possible to return every word when the number of time steps is too small. This approach allows the use of different sized images and in principles allows the usage of images which are bigger than all training images. One problem that needs to be solved with this approach is that there are possibly more time steps than the length of the character sequence in the image. Also, the alignment between the input windows and output characters is not clear because multiple windows may contain the same character or one window might contain multiple characters. This can be solved by using the CTC loss as it allows non-fixed alignments and to run for more time steps than the length of the label sequence. To support the use of the CTC loss an additional output character, the *blank* character, is added to the Softmax. During training the probability of the correct character sequence is maximized. At inference time, the character sequence with the highest probability is used as output, which is found by using beam search. Each image in a batch must be of the same size and each label sequence length must also be equal. For this reason, padding is performed on both. To reduce the padding, bucketing with four different buckets is used. In contrast to the Encoding NN, bucket is performed on the width of the input images, since this will decide for how many steps the RNN runs. It also influences the cost of the convolution operations as those are more expensive for bigger images. The total amount of padding can also be reduced this way because, images must be padded much more than sequences, since the vectors in sequences are significantly smaller. This has a major impact on the memory that must be transferred between the host system and the GPU and reduces therefore the training time.

4.2.4. Sliding Window Attention Neural Network

The Sliding Window Attention Neural Network augments the Sliding Window NN by a soft attention mechanism. The attention mechanism is applied on the output of the CNN. The attention mechanism is calculated using the previous attention map, the previous state

and the current input. It is therefore a hybrid version of the content based attention and of the location based attention mechanism. This type of attention allows to make use of similarity in time and take the current information in the image better into account. It seems beneficial to use windows which overlap because otherwise location based attention might not be as useful because subsequent images might have nothing in common. It might be hard to estimate a good next attention position, when the content in the image is only known at this time step. If the windows overlap subsequent windows have more correlation to each other. For this reason, the model is trained in all cases with overlapping windows. The actual model implements attention using the state of the first layer of GRU cells, the output of the CNN, of the current window, and the last attention map (Figure 4.10). The calculation of the attention map is inspired by [50] and performed in the following way:

$$\boldsymbol{e}_{i,j} = \boldsymbol{v}_a \tanh(\boldsymbol{W}_a \boldsymbol{s}_{i-1} + \boldsymbol{U}_a \boldsymbol{h} + \boldsymbol{V}_a \boldsymbol{\alpha}_{i-1}) \qquad (4.1)$$

Here $\boldsymbol{v}_a$, $\boldsymbol{W}_a$, $\boldsymbol{U}_a$, $\boldsymbol{V}_a$ represents the trainable parameters of the attention feed-forward NN. The size of the attention map is equal to the width and height of the output of the CNN, given some input window. Each feature map, in the result of the CNN, is element-

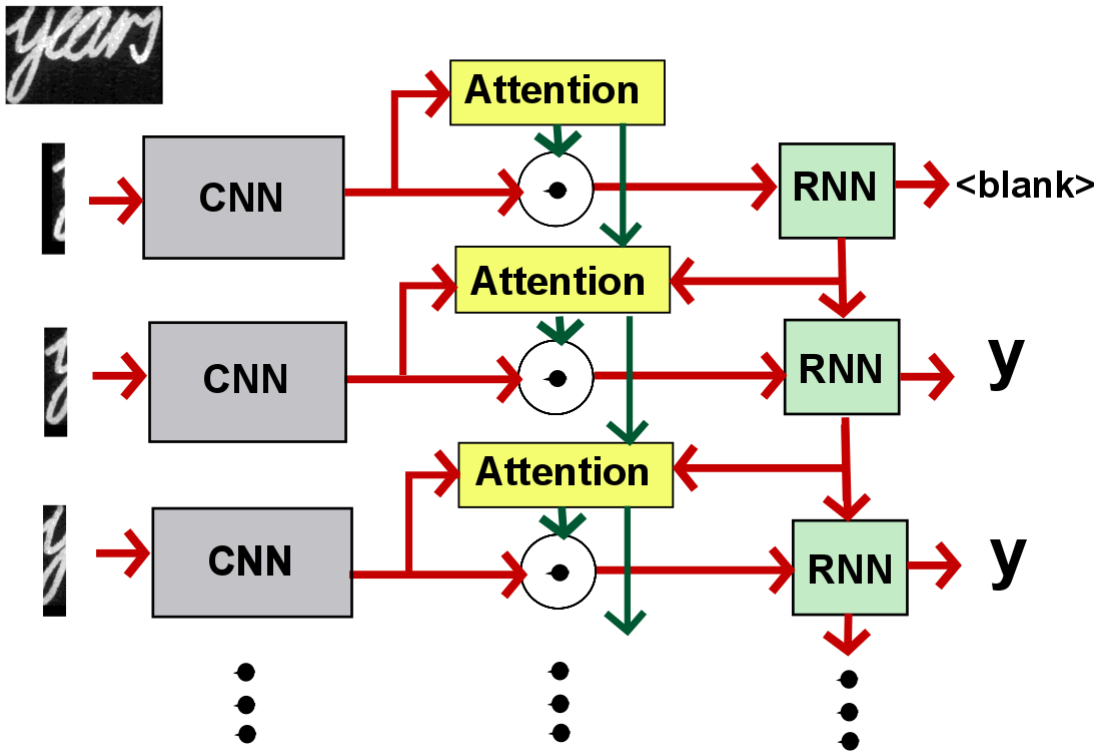

Figure 4.10 Sliding Window Attention Architecture [80]

wise multiplied with the attention map and then the sum is taken. Therefore, for each feature map one number is generated. The resulting vector is feed into the RNN as input.

The NN stays otherwise the same as the Sliding Window NN. An additional adjustment was made for the last model. The state of the first layer directly influences the resulting attention map ((4.1), which might be a problem because the state of the RNN is initialized to zero for all used models. It has therefore no influence on the attention map. The input from the previous attention map is also zero, because there was no previous attention map at the first step. Therefore, only the content part is used to calculate the attention map. This might influence the resulting accuracy of the system in a non-beneficial way because the weight matrix $\boldsymbol{U}_a$ is trained for two different situations. The situations are the regular case where it is not the only influence on the attention map and the case at step one where it determines the attention map alone. To reduce the influence of this the first RNN layer uses information from the input to initialize the start state as was done in [52]. The state is initialized in the following way using only the output of the CNN applied to the first input window:

$$\boldsymbol{s}_0 = \boldsymbol{W}_{init}\boldsymbol{o}_{CNN} + \boldsymbol{b}_{init} \tag{4.2}$$

$\boldsymbol{W}_{init}, \boldsymbol{b}_{init}$ are trainable parameters that can be learned by the NN and $\boldsymbol{o}_{CNN}$ is the output of the CNN. The full image or a down scaled version could have also been used, to initialize the state. This would require a possibly expensive matrix multiplication and additional padding which would defy the purpose of this architecture. On the other hand, this might introduce global information, about the image, to the RNN.

4.2.5. Parameters of the performed Training

The NNs are trained using Tensorflow version 1.1 [70], using the python API [82]. The

Conv. Layer 1 parameters	5×5×1 ×48	Num. RNN Layers	3	Adam: ε	10^{-8}
Conv. Layer 2 parameters	5×5×48 ×64	RNN Cells per Layer	512	Adam: β_1	0.9
Conv. Layer 3 parameters	5×5×64 ×128	Adam: α	10^{-4}	Adam: β_2	0.999

Table 4.1 Shared Parameters

training was performed using a NVIDIA GeForce GTX 1060. The benefit of using Tensorflow is that it allows the use of the highly optimized CuDNN library version 5.1 [66] [83], for NVIDIA GPUs, and allowed the creation and training of models while the framework was still in development. To use CuDNN, CUDA with toolkit version 8.0 was necessary.

The base parameters for all NN are list in Table 4.1 (Those parameters where used when nothing else was stated). The baseline NN was trained only once using the parameters listed in Table 5.2. For the sliding window NN multiple networks, with different parameters, were trained. A change in parameters allowed an analysis of the influence, on the performance of the NN, they have. This allows better guesses on how a NN must be designed and trained to perform well on the IAM data set. Four sliding window NNs were trained with the parameters stated in Table 5.4. The sliding window attention NN was trained three times with the parameters stated in Table 5.5.

5. Results

This chapter will start by presenting an evaluation of the framework followed by an analysis of the trained Neural Network models.

5.1. Evaluation of the Framework

The implemented Framework needs to be evaluated. For this reason, this chapter will begin with a general analysis of the Framework, followed by an Evaluation of the implemented Convolution Kernel.

5.1.1. Evaluation of core elements

The developed Framework was successfully applied to different Neural Networks architectures. It was used to train a CNN on the MNIST dataset where it reached a precision of 99.05%. Additionally, it was trained to solve the MNIST task using a RNN, where the rows of the image are the inputs for the different time steps. There it reached a precision of 97.7%. At the end, it was applied on so called “Kassentrenblätter” which will be explained in a following section. The major requirements the Framework should fulfill are a reasonable performance, easy extensibility, ease of use in creation of new models and cross-platform compatibility. The cross-platform compatibility is fulfilled by using OpenCL 1.2 which is available for many devices. One possible problem is the use of functionalities that only exist since C++ 11. This might pose a problem because the standard is not necessarily support by all relevant compilers. The execution of the Neural Network consists mainly of executing kernel function and iterating over an array. The

execution time of the training depends, therefore majorly on the implemented kernel functions. An except to this is the fact that RNNs are unrolled statically. It is still possible to run the RNN using varying sequence sizes but it requires much padding and that all gradients/results, after the last time step are set to zero. This can be made easier by using a loss function, that only computes the error/gradient when the current label is no padding. This might pose a problem when the sequence lengths vary greatly between examples and the maximal sequence length is much longer than it is for most examples. The RNN must run for the maximal number of time steps, independently of the current sequence length. One situation in which this might pose a problem is handwritten character recognition. In handwritten character recognition, the longest words may consist of 15 or more characters, while there are many words, like the word “you”, that are much shorter. At the moment, the size of each input image must be the same, which can lead to inefficiencies. The convolutions are performed on padded images, but it would be more efficient if the convolutions would be performed on smaller images and padded afterwards. A new model can be implemented by adding operations and buffer to a graph which seemed to be relatively direct. This is similar to how some other Deep Learning Frameworks [70] [68] create its models. One critical point may be the fact that the implemented operations are relatively fine grained and one layer is often constructed out of multiple operations. The bias and activation functions are for example single operations which must be created separately to generate a complete layer. A typical convolution layer would need a convolution, a bias and a ReLU operation. This could easily be solved by adding functions which create high level layers automatically using multiple more fine-grained operations. The GRU layer is implemented this way, because it is computed by multiple more fine-grained operations. While it is relative easy to add new operations, it can still be quite exhausting when it needs to be done to often. It is also necessary to implement new kernels, when the implemented kernels do not fulfill the needs. For each new operation, a new class must be implemented, that must implement the Instantiate function. The function requires the creation of a Tuple for each hardware operation. This can look quite confusing due to the number of necessary Template arguments. To conform to the general conventions a function for the creation of the Operation must be implemented in the OpManager class, but this is not mandatory. Even so it is still relatively flexible to create operations which take a multitude of different numbers of arguments and types.

5.1.2. Evaluation of the Convolution Kernel

The convolution operation is, besides the matrix multiplication, one of the most expensive operations in deep learning. A special OpenCL kernel was implemented for it. An analysis in many different situations is necessary, since, in Deep Learning, convolutions are used in a wide range of settings. The performance of the kernel is also compared to another approach for implementing the convolution, which is based on [84]. The convolution, used for comparison, uses a Tile based approach, where a tile of the image is load into local memory. A complete filter image of the filter volume is loaded into local memory as well. The approach was extended to incorporate the third dimension of the filter volume, multiple different filters and that the input is gathered into a batch. The additional dimension of the filter was accounted for by looping over each image in the input volume. The higher number of filters and image, at the same time, was accounted for by using the z direction of the work group. All elements in a local work group have the same image tile loaded but for each different z is another filter loaded. Therefore, the loaded image tile is used for different filter results increasing the data reuse. The z direction also accounts for different batch elements, by using multiple work groups in z direction, for all filters, for each element in the batch. Therefore, a higher number of elements in a batch increase the degree of parallelization. The convolution kernel evaluation was performed on an NVIDIA GTX 1060 with NVIDIA driver version 382.33. The kernels were evaluated by fixing all parameters to a default configuration (T) and varying one of them. The work group sizes and the tile width/height are equal for those kernels and are

Rows per Load	Tile Width	Tile Height	Image Width	Image Height	Batch Size	Input Filter Depth	Output Filter Depth	Filter Size	Stride
15	16	16	240	240	1	8	44	5	1

Table 5.1 Evaluation Parameters

therefore used interchangeably. The execution time of the kernel is measured to evaluated the parameter settings. No OpenCL compiler optimizations like fast relaxed math or “mad” optimizations were enabled. The rows per load parameter specifies the number of rows of the matrix, that are created by the algorithm. It must always be a multiple of the filter width and smaller than the tile width. The tile width specifies the number of columns, that are loaded into local memory at each loop step. It therefore specifies the number of output pixels, that are calculated. The tile height specifies the number of different filters, that are loaded at a time. The filters used in this example are always symmetrical, therefore a filter size of 5 specifies a 5×5 filter. The same holds for the

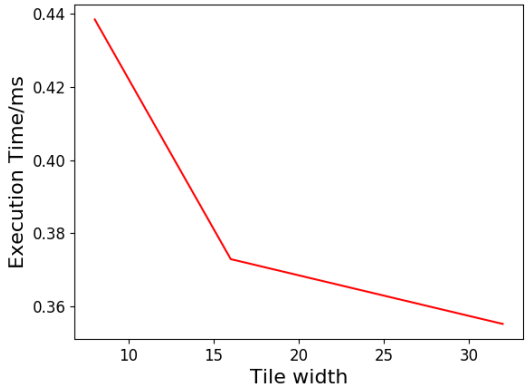

Figure 5.1 Change in Tile Width

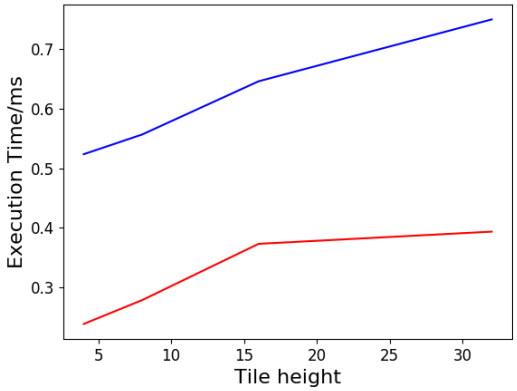

Figure 5.2 Change in Tile Height – (Blue) 20 Filter, (Red) 44 Filter

stride. Padding is always adjusted to be equal to the half width of the filter size, rounded down. All the parameters in the table are varied once to evaluated the influence of them on the performance, except the rows per load, which is always better when it is higher. The First three parameters, rows per load, tile width, tile height, are in principle independent of the convolution, that is performed and allow the optimization of the convolution operation. The results of varying the tile width and tile height can be found in Figure 5.1 and Figure 5.2. In the Tile width example, the rows per load parameter is also adjusted, because for a tile width of 5 it is only possible to load 5 rows. For a tile width of 32, the rows per load parameter it set to 25, allowing maximal performance. The tile height was varied using two different numbers of output filters. The reason for this is, that the output depth could have an influence on the behavior of the performance when it is higher. The output depths are 20 and 44. As can be seen a higher tile width improves the execution time while a higher tile height reduces the performance. The behavior of the execution time for different tile heights seems to stay the same for different numbers of output filter depths. The behavior of the execution time with respect to the tile width is not unexpected, because higher widths allow more data to be loaded at the same time and the loaded kernel to be used for more image pixels. It also allows a higher number of matrix rows to be loaded into local memory at the same time. A higher tile height decreases the execution time probably, because not all threads are used to load the image tile and stay idle, therefore. Only a limited number of elements must be loaded at the same time and only a few threads perform some work. Also, the number of outputs filters is relatively small so there might be relatively many threads that perform no real work, because they are out of bounds of the kernel depth. Using a tile height of 16 and a filter

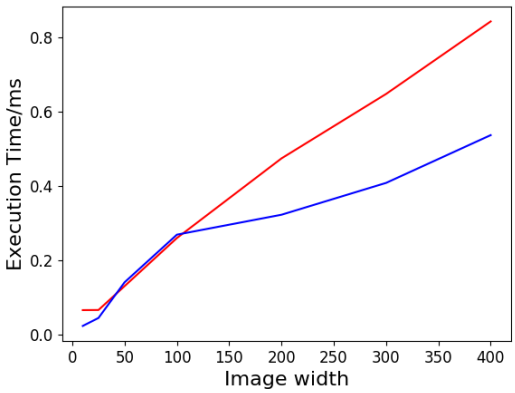

Figure 5.3 Change in Image Width – (Red) Transformation based, (Blue) Tile based

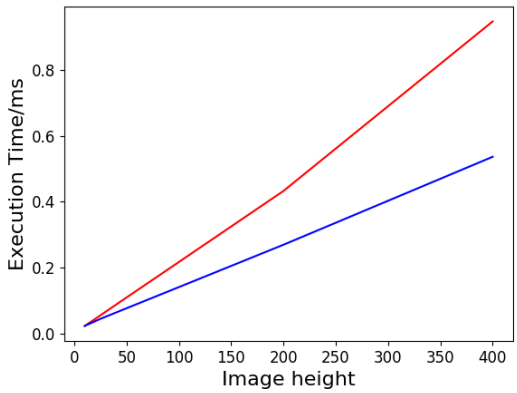

Figure 5.4 Change in Image Height – (Red) Transformation based, (Blue) Tile based

output depth of 20 results in 8 threads, that perform no work and just stay idle. The next parameters, that are varied are the image width and the image height. The results can be seen in Figure 5.3 and Figure 5.4. For the height exists a linear dependency for both implementations of the convolution kernel. This behavior was expected because the work, that needs to be performed is linear in the image height (quadratic in height times width). The slope is different for both implementations of the convolution operations. The transformation based convolution performs worse than the tile based implementation. A possible reason is that the computation for each pixel is more expensive for the transformation based implementation. The problem is probably the restriction on the size of the tile of the matrix, which is generated by the algorithm. The tile size in y dimension is determined by the rows that can be loaded at each step, but this factor must be smaller than the tile size in x direction, otherwise not enough elements of the kernel can be loaded. Many threads might stay idle, when the image tile is loaded into local memory. To reduce this, it would be beneficial to let each thread load multiple parts of the kernel into local memory, thereby allowing a higher number of rows to be loaded at the same time. The execution time of the transformation based kernel is also linear in the width, except for very small widths. In the region of small widths, the work group is bigger in x direction, than the width and not all threads perform computations. Some threads stay idle. A small increase in the width, therefore only achieves that less threads in the work group perform no work. It does not change the total execution time of this work group. The tile based convolution behaves relatively similar at the beginning and enters than a region, where it behaves more like the height situation, but with a lesser slope. In both cases the transformation based convolution performs worse, which is probably due to the before

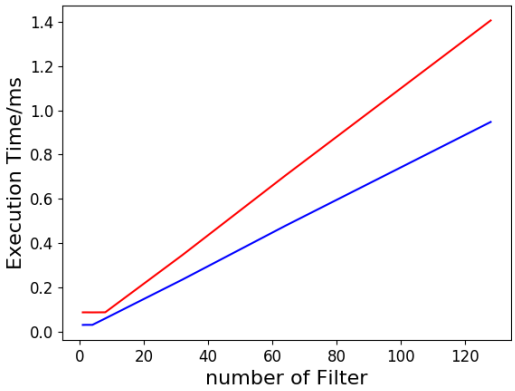

Figure 5.5 Change in the Number of Filters – (Red) Transformation based, (Blue) Tile based

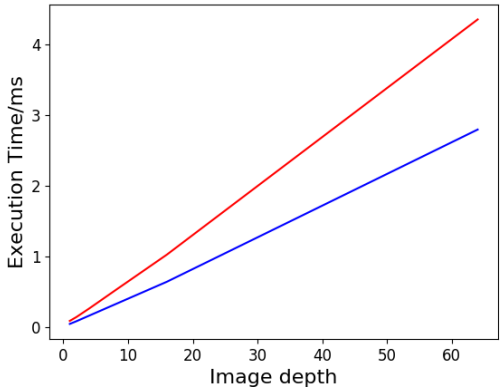

Figure 5.6 Change in Image Depth – (Red) Transformation based, (Blue) Tile based

mentioned reason. The execution time for each kernel with respect to the input depth and output depth can be seen in Figure 5.5 and Figure 5.6. In both cases, both implementations exhibit a linear increase. The slope is again smaller for the tile based convolution, which is probably again due to the idle threads while loading. The execution time, with respect to the output depth, starts again with a flat surface. The flat surface is caused by the fact, that the work group size in y direction is higher than the number of filters, which are read in parallel by one work group. The flat region is smaller for the tile based approach because the work group size is smaller. The last two variable influences that are analyzed are the execution with respect to the size of the filter and the stride (Figure 5.7 and Figure 5.8). The execution time decreases near quadratically, for the transformation convolution, with respect to the stride. This is to be expected because the stride is adapted for both dimensions, therefore the output width and height decrease in a linear fashion. The tile based implementation does not decay quadratically and even has a peak at a stride of 4. This behavior is due to the fact, that more elements must be loaded, than there are work items in the local work group. Therefore, two iterations for loading parts of the image must be performed, which increases the execution time quite much. The transformation convolution performs better for higher strides than the tile based convolution due to the increased number of loads each thread must be perform. The increase in loads rises quadratically for the tile based convolution, whereas the increase for the transformation based convolution is linear. The filter size increases the execution time near quadratically for both implementations but the tile based implementation suffers again from a higher slope. The quadratic dependence is also to be expected because the convolution depends

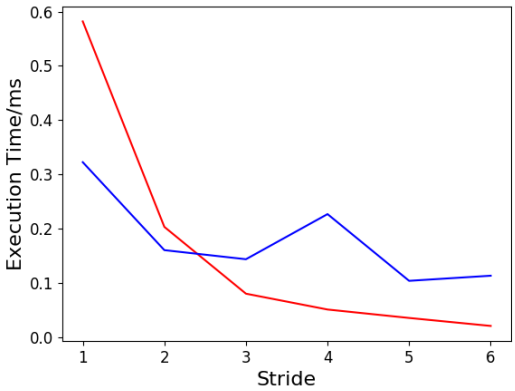

Figure 5.7 Change in Stride – (Red) Transformation based, (Blue) Tile based

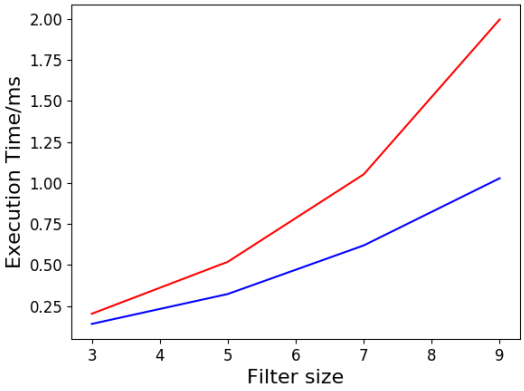

Figure 5.8 Change in Filter Size – (Red) Transformation based, (Blue) Tile based

quadratically on the filter size if the kernel is symmetrical. In the current state, the transformation implementation performs worse than the tile based convolution. It also has some problems, with respect to the possible parameters, because some parameter configuration may lead to errors. The rows per load parameter is restricted and may result in errors if the parameter is chosen in a wrong way. The parameter has also a major influence on the performance of the algorithm. In order to increase the performance further the threads might need to be able to load more data points at the same time, therefore archiving a more evenly distributed workload per thread, while loading.

5.2. Evaluation of the Neural Networks

In this chapter, the results of the different Neural Network architectures will be presented and analyzed. The results will be used to analyze the behavior and problems of the NNs in order to improve them in future works. At the end, a comparison between the different NNs and the state of the art results in handwritten character recognition using deep learning will be made.

5.2.1. Evaluation of the Encoder Decoder Neural Network

The Encoder Decoder Neural Network was trained only with one configuration because its primary purpose is being a baseline to compare against for the other NNs. One additional reason only one version was trained, is that the most meaningful change, an increase in the number of cells in the RNN, could not be made due to hardware limitations. The reason for this is the matrix that needs to be of size, with the number of cells in the RNN and the depth, height, and width of the output of the CNN. The depth, height and width are quite big because the input image has a height of 241, a width of

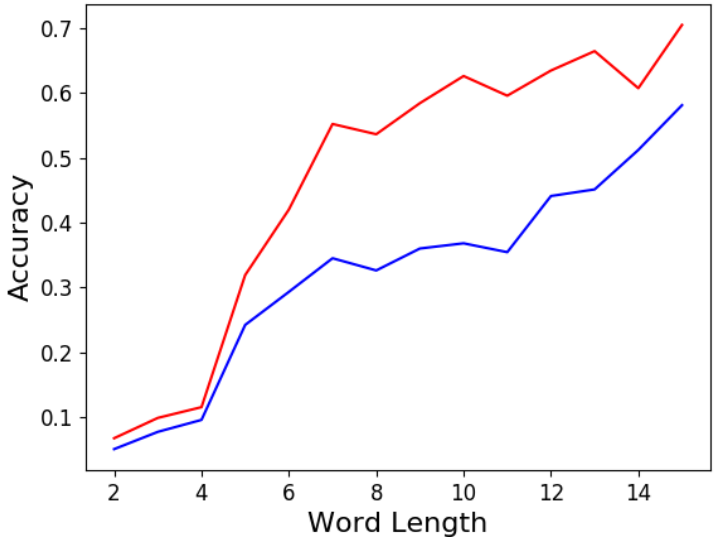

Figure 5.9 Encoder Decoder Accuracy relative to the Word Length – (Red) Correct Input, (Blue) Last Output as Input

1087 and the output depth is 128. This makes the matrix rather big even though the resulting RNN is relatively simple and not very deep, which is a big disadvantage of this NN. The results of the NN can be found in Table 5.2

Name	Num. RNN Layers	RNN Cells per Layer	Accuracy(CER) in %
NN one	1	256	29.2

Table 5.2 Results and Parameters of the Encoder Decoder NN

The NN archived a total CER of 29.2% on the test set. Because only one version of this model was trained no direct comparison to other versions of it can be made. Looking at the error distribution relative to the word length and especially what types of errors are common, reveals quite some information about the problems of this model. The distribution of errors relative to the word length in Figure 5.9 shows, that the performance degrades heavily as the words grow longer. For words, that have a length of three and shorter the NN can archive an CER of 11.5% and better. This is followed by a decline of the CER from 31.8%, for three characters, to an CER of roughly 70.4%, for words with a length of 15. This indicates that the NN was not really able to learn to recognize long words from images. A look at the results for different words shows a reason for the high errors occurring for long words. The mistakes the NN made had a rather special structure to them, as can be in seen in Table 5.3. It lists the correct word with a white background and the word that was returned by the NN with a grey background. As can be seen most words, that were returned by the NN are real words or at least something relatively close

Label	Ret. Word	Label	Ret. Word	Label	Ret. Word
years	young	people	resource	breakfast-nook	indifferenteal
asked	about	far	for	Courtenay	Government
most	meet	immediate	remember	friendship	fillowalds

Table 5.3 Wrong Word examples

to them. In many cases the by the NN returned words are not the words that were present in the image. It seems to have learned a model of the words in the data. The returned words seem to be biased on the characters of the word in the image, especially the first character and on the length of the word. An example is the word “asked” for which the NN returned the word “about”. They have the same length and the same first character but are completely different otherwise. Another example is the word “breakfast-nook” and the returned word “indifferenteal”. The NN put more importance on having a word of the same length, than to return a real word. “indifferenteal” is no real word but close to “indifferent”. The total number of characters in both words is the same but they are completely different otherwise. One possible reason for this behavior could be the memory capacity of the RNN, since it only has 256 cells to store information over multiple time steps. It must store all necessary information in its state to recognizing the word correctly because the image is only passed to it once at the beginning. This information can potentially be too much for the state, therefore a selection of what information to store must be made, additionally it might be necessary to store some intermediate information which makes the memory even more a problem. A higher number of cells might therefore, help to improve the results by giving it more capacity to store information extracted from the image. Since the input to the NN is the last output and it might have not enough memory to store all information, it might have learned to predict the most probable word given the information from each input and the information stored in the state at initialization. The named reasons might explain why it is able to read short words quite well, while it has much more problem when the word grows longer. For short words, not so much information is needs to be stored in the state compared to a situation of longer words. The information for shorter words must not be kept as long in the memory as the information for longer words because all outputs are generated relatively early. Longer words require some outputs to be made at a much later time step therefore, requiring the storage of more information longer. Another reason why the NN reaches a higher accuracy for short words might be the fact, that the number of different short words is significantly lower than the number of different words with more than three

characters. The train set contained 194 different words that are shorter than a length of four while there are much more different words for each other length alone. Therefore, the probability of returning the right word for a short length is much higher because there are less words, assuming it has learned a word model. Also, if the NN gets the first character right the number of possible different next characters is quite short, for words with a length shorter of four. There are much more different combinations for words that are longer. If the NN makes the mistake and outputs the wrong word, but has the first character right, then the CER is in most cases quite high. For short words is the CER not as high, when the NN returns the first character correctly, because the CER is normalized by the length of the word. Additionally, the NN is often able to predict the length of the word correctly and therefore, returns the correct last character which is always the "*eow*" character. This also reduces the CER for short words drastically, because the CER is normalized by the word length. Another difference exists between the accuracy of the training and the validation test, which was roughly around 5%. Even though such a difference was to be expected, it already happened when samples from the training set were passed the first time to the NN. This is unexpected, because in the first pass (epoch) through the training set all examples are seen for the first time. The reason for this is the fact, that at training time the correct label, which should have been returned is passed as input to the RNN. At test time, the last returned character is passed to it. To prove this the Encoder Decoder model was again evaluated on the test set but this time the correct label is passed to it instead. This change decreased the CER by 9.3% to an CER of 19.9%. This shows that the difference indeed resulted from the different situation at training and testing. The behavior of the NN changes in some respects as well. The accuracy decreases in a similar way to befor when the word length is increased but the CER value for each length is smaller in general. This can be seen in Figure 5.9. The types of errors the system does change in this setting. Before the NN tended to return a word that existed or was at least close to a real word. In the new setting, the returned words tend to be often no real words or as close to it. The reason for this might be the fact, that the inputs passed to it conflict with the information stored in the state, since there is a discrepancy between the input and the last output. This increases the correctness because it possibly biases the word in the correct direction and leads it away from wrong words. The change increasing the chance for it to output a correct character and reduces the CER, thereby. Even though the change increases the total performance of the system it is not a viable option in practice. It needs the correct character for each time step to be known beforehand, therefore, contradicting the purpose of handwritten character recognition.

Name	Num. RNN Layers	RNN Cells per Layer	Window width	Window offset	Accuracy(CER) in %
NN one	3	512	16	16	19.9
NN two	1	512	16	16	21.1
NN three	3	128	16	16	25.8
NN four	3	512	24	10	11.4

Table 5.4 Results and Parameters of the Sliding Window NN

5.2.2. Evaluation of the Sliding Window Neural Network

The Sliding Window Neural Network was trained using four different combinations of hyperparameters, therefore, allowing an analysis of the influence of the parameters on the result. The used parameters, that were varied can be seen in Table 5.4. To address them separately the NNs are referred to by NN one to four or first to fourth NN. The last entry in Table 5.4 Results and Parameters of the Sliding Window NN displays the performance of each NN on the test set. All of them archived a different CER on the test sets. The first one archived a CER of 19.9%, the second archived a CER of 21.2%, the third of them an CER of 25.8% and the fourth one a CER of 11,4%. The first thing apparent in those results is that the fourth neural network archived the best result, by a margin of 8.5%. The difference between the fourth and first one is that the window is bigger and the stride is smaller than the window with, by 14 pixels. The windows overlap, which is not the case for the three other NNs. Looking at the remaining three NNs one can see that NN one and two differ only by 1.3 % whereas the difference between the second and the third is 4.6%. Each window for the first three NNs contained, in most cases, only a part of a character or parts of two characters that intersected. This means that the NN might have to store information from one character over multiple time steps to recognize the given character correctly. Additionally, it might be beneficial to store information from previous characters allowing it to interpret the following characters differently. There are examples of words where a single character can't be correctly classified if only the image information of this character is used. Some characters might be written in a way identically or very similar to another one or a correction of a character might happen, so that it is not clear which character is displayed. Examples are images of the word "Egypt" and of the word "knew" in Figure 5.10.

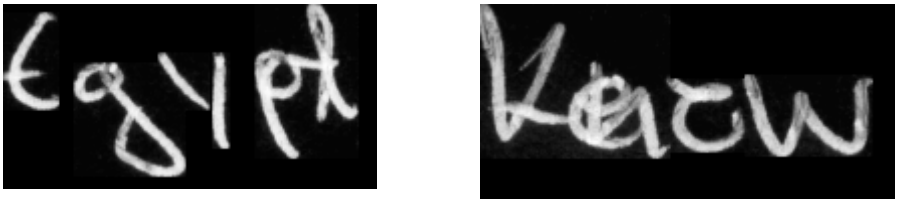

Figure 5.10 Hard Characters – (Left) The E in Egypt is hard, (Right) The n in knew is hard [80]

Both words have a character which is hard to read when only the image information of the character is used. In the image of "Egypt" the character E could also be read as t. The "n" in knew was corrected, so that it is hard to read the correct character if only the image information of this character is used. For the before mentioned reasons it might be practical to have more capacity to store information. The information is stored in the state of the RNN, therefore, an increase in the number of cells and layers might allow the RNN to store more information. This is reflected in the results. The NNs with the highest number of cells and layers archive the best results (NN one and four). On the other hand, NN two and three don't differ much in the total number of cells they have in total but the accuracy of them differs quite much. NN two has 512 cells in one GRU layer and NN three has 384 cells split over three GRU layers. The other two NNs have 1536 cells in total. This hints at the fact that a bigger first RNN layer might be more beneficial than having many smaller RNN layers. A possible reason for this might be the amount of information that can be transferred from one RNN layer to the next. The outputs of LSTMs and GRUs have the same size as the number of cells, therefore, the first layer must encode the information from its input and its state to transmit information to the second layer. The first layer receives all information from the input and must decide which information it itself stores and which information should be passed to the next layer. The next layer only receives the reduced/filtered information. All of this depends on the number of cells in the layer, so that a small number of cells requires the layer to ignore more information. This could be especially important, because the output of the CNN is very high dimensional and a very strong compression between input and output is necessary, therefore. This could explain why the third NN has a lower accuracy while the first two archive more similar results. While this explains the differences between the third one and the rest, it does not explain why the fourth one is still so much better than the rest. The only difference between the first and the third NN is the window size and offset. It could be useful to take a closer look on the windows that are generated. An example is Figure 5.11, which contains the word "years", inclusive padding, and a few of the resulting windows for the first and fourth NN. For this example, the first NN returns

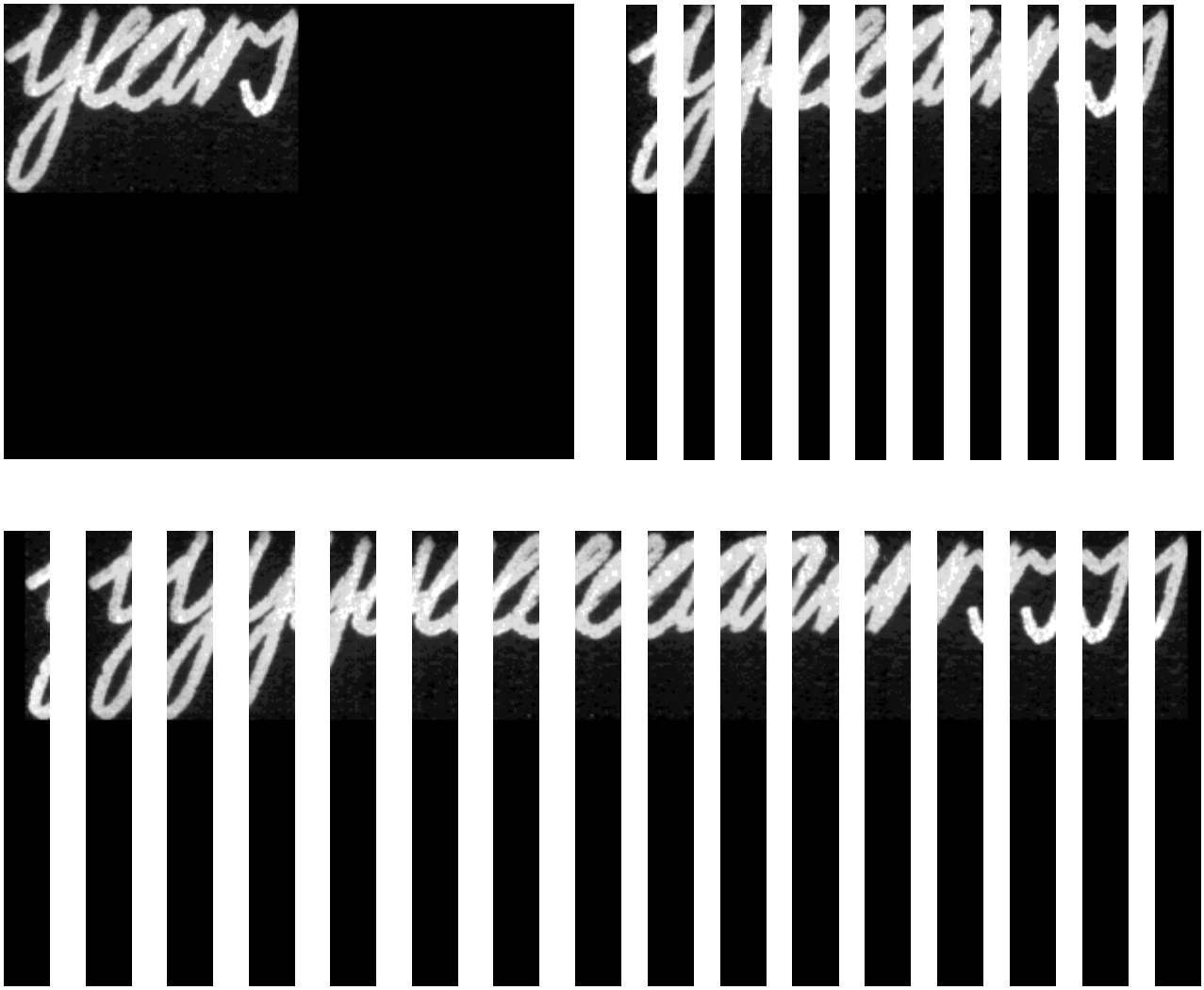

Figure 5.11 Example Windows – (Top Left) Full image, (Top Right) window offset 16 and window width 16, (Bottom) window offset 10 and window width 24 [80]

the word “gter” whereas the fourth NN is able to output the correct word “years”. One reason for why the first NN outputs an “g” and “t” at the beginning might be, because the second window can be interpreted as an “g” followed by the character “t” in window three. The first three NNs might have the problem with deciding when a character starts and when it ends. The windows might be too small and the NNs might not have stored all necessary information contained in the windows. The fourth NN on the other hand does not need to store so much information about the windows, because they overlap and it receives more information in total. It also has more time for each character, because the offset is with a value of 10 pixels smaller compared to 16 pixels. One additional benefit might be that some character almost completely fit into one window. The characters “y”, “s” or “a” almost completely fit into one window, leaving only a small part at the beginning and end out. Because the stride is quite small it is probable that a window exists that aligns reasonable well with each character, if they fit into one window. This is not so often the case for bigger offsets and smaller windows, because in this case characters are

often split up between multiple frames. Only very small characters have a higher probability of fitting fully into one window. The character y is an example for a character that was split up into an unfortunate way, so that the NN recognized two different characters. Looking at words that were recognized entirely wrong by the fourth NN supports this assumption because the characters in the images were written rather big, so that they did not fit into one window alone.

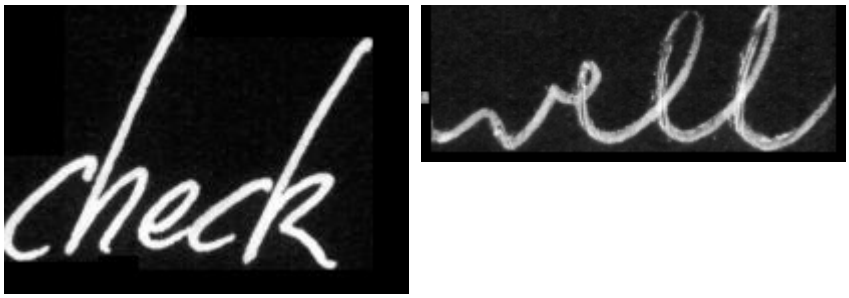

Figure 5.12 Big Character – (Left) check, (Right) well [80]

The characters in Figure 5.12 left, of the word “check” are written rather clearly but the fourth NN returns the word “dade”, which is entirely wrong. It could be because the fourth NN recognized both c in combination with the long vertical line of the character “h” as “d”. The word “well” in Figure 5.12 right was recognized as “isvel”. The first part was especially wrong, which is also the part, where a character is split up the most over multiple frames. The aforementioned reasons might explain the difference between the

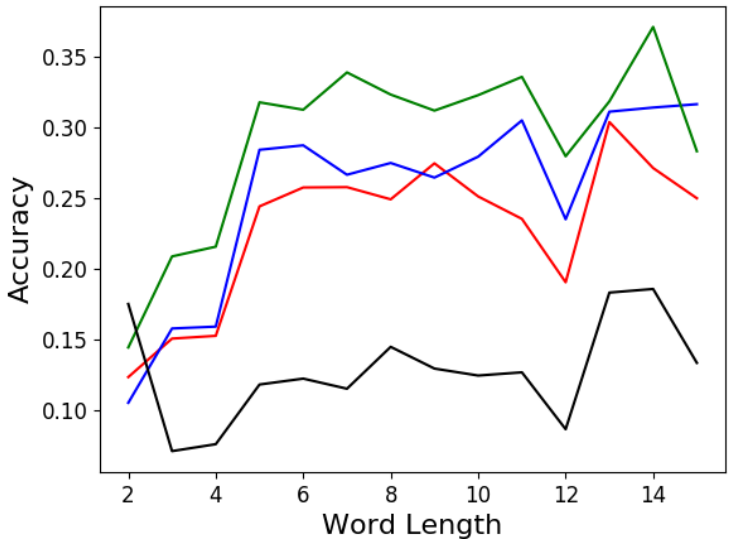

Figure 5.13 Sliding Window Accuracy relative to the Word Length – (Red) NN One, (Blue) NN Two, (Green) NN Three, (Black) NN Four

first and fourth NN but more experiments are necessary, since different values of window offsets and sizes might still archive better results. The results also fit together with the

results the first three NNs, because in this case the capacity in the first RNN layer was a crucial factor in the performance of the RNN. Since Each character is split up over multiple frames, the RNN needs to store more information over multiple time steps therefore requiring a higher capacity.

The test set contained images of words with many different lengths, therefore the CER was evaluated on all different available word lengths for each model. The results can be seen in Figure 5.13. It is important to remember that to each original word label an end of word token is added, therefore the shortest word has length two. The average error rises roughly with the length of the word for each NN, meaning that all NNs are better at short words while long words tend to be a little bit harder for them. The training set contained more short words and less long words, so that training was done more often on shorter words. The distribution of word lengths in the training set could, therefore, have an influence on the results for different lengths of words. Especially the number of combinations of characters for short words is less than the number of combinations of characters of long words, therefore, training would be performed more often on those character transitions. To analyze this more different data sets for training would be useful.

Figure 5.14 Uppercase was [80]

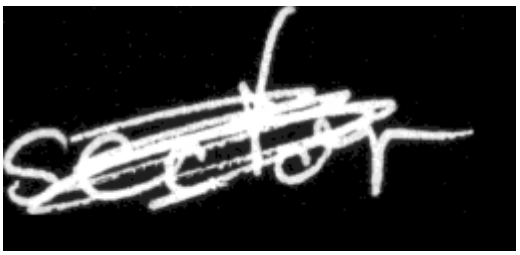

Figure 5.15 Crossed Over sector [80]

The IAM dataset contained the "#", which has a special meaning. The words in the images in the data set that were labeled with this character were crossed over. In this case the task was not to recognize the written word but the fact that is was crossed over. The word in Figure 5.15 was recognized wrong by all four neural networks. The NNs tended to return parts of the crossed over words. They therefore were not really able to learn what it means when the words were crossed over. this might be a problem with the approach, because the NN has only local information from the current and all windows that came before, available. The character "s" in Figure 5.15 could be a problem because the crossing does not really start before the end of this character. The NN makes a mistake when it outputs the character "s" directly when it is encountered. Information from a later part of the image are required to understand, that the character "s" is wrong. It could be

necessary to provide the RNN with more global information from the image. Another class of errors are differences in upper and lower characters. There characters, that can be written in a similar way when written by hand, for example “w” and “W”. The word “WAS” in Figure 5.14 was recognized as “wa” by the fourth NN. The problem is that the characters in the data set have many different sizes and especially the first window does not contain information about the scale of the complete word. There is no information on how big the first character is relative to the other characters. This again may be, because it has only local information available. A similar type of mistake was the character “”” because it was often mistaken for a “,” and the other way around. The CNN discards some local information by reducing the resolution and max-pooling pixels together into one. It might not be clear if the character was on a lower or higher position, which is a major distinction between both characters.

5.2.3. Evaluation of the Sliding Window Attention Neural Network

The Sliding Window Attention Neural Network was trained using three different hyperparameter settings. An analysis of the influence of those on the CER will be made and the results will be discussed. The parameters, that were varied and the corresponding results are presented in Table 5.5

Name	Conv. Layer 1 parameters	Conv. Layer 2 parameters	Conv. Layer 3 parameters	State Init.	Accuracy(CER) in %
NN one	5×5×1×32	5×5×32×48	5×5×48×64	Zero	15,0
NN two	5×5×1×48	5×5×48×64	5×5×64×128	Zero	14,0
NN three	5×5×1×48	5×5×48×64	5×5×64×128	First Window	13,1

Table 5.5 Results and Parameters of the Sliding Window Attention NN

The NNs in this section will be named as in the table to address and enumerated them by their position in the table. The CERs on the test set are quite similar for all of them. The first NN archives an CER of 15%, the second NN an CER of 14% and the third NN an CER of 13.1%. All of them are roughly one percent apart. While the results of all of them are quite close to each other, they show different behaviors when the average CER for each word length is focused (Figure 5.13). The first two NNs show a slight accuracy reduction, when the length of the word increases. A more consistent behavior over a wide range of different word length is shown by the third NN. The third NN archives the best results for word lengths of two to four, followed by an increase to an CER of roughly14-16%. An exception are the words of length 12, for which each NN archives better results

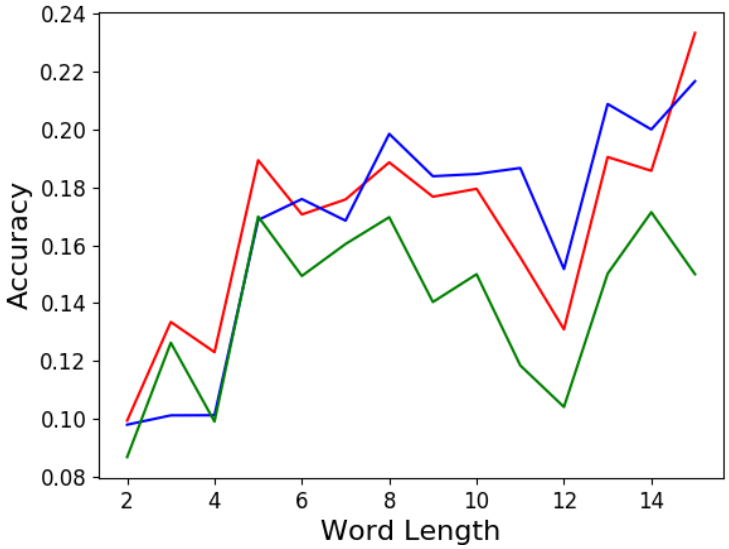

Figure 5.16 Sliding Window Attention Accuracy relative to the Word Length – (Red) NN One, (Blue) NN Two, (Green) NN Three

than for the length 11 and 13. The reason for this could possibly be the fact, that most images of words with length 12 in the test set are written rather clearly, an example being the word “Christopher” in Figure 5.17. Each character in this word is written rather clearly

Figure 5.17 Chrisotpher Example [80]

and the characters are already somewhat separated from each other. Many words with length 12 in the test set are comparable to this one. The reason for the difference of the behavior between the first two NNs and the third NN must still be found. One reason for this might be the fact, that the initialization of the state with zero makes it harder for the RNN to store information. The third NN on the other hand, can adjust the start state with a parameter matrix, which can be learned. Therefore, it can learn to initialize the state in a way that makes it easier to store useful information. To understand this more clearly a look at the attention map, which is generated by the NN might help, but to do this, the attention map needs to be transformed back, since it is generated for the images that result from the CNN. As the CNN performs three convolutions, each followed by a max pooling operation. The attention map is first scaled up to reverse the effects of the max pooling and then convolved with a 5×5 filter to reverse the effect of the convolution. The scaling

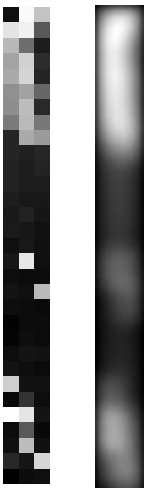

Figure 5.18 Attention Map

up is performed by duplicating each pixel in both directions. The filter in the convolutions consist only of ones, because the goal is to know on which places the NN puts its attention and not how the pixel influence the resulting feature map. The results of this transformation can be seen in Figure 5.18 (the image of the original attention map was scaled because it is otherwise to small). As one can see the result of the transformation is mainly a blurring of the original feature map and an increase at positions where many pixels get a high value in the attention map. Figure 5.18 shows the first four resulting attention maps for the image of the word pleasant for all three NNs. There is a clear difference between the attention maps of the first two NNs and the attention maps of the third one. The first two NNs don't really focus on one region but pay relatively equal attention to all places in the image. Even though the biggest part of the image contains no content and only exists, because the image height needs to be the same for all images.

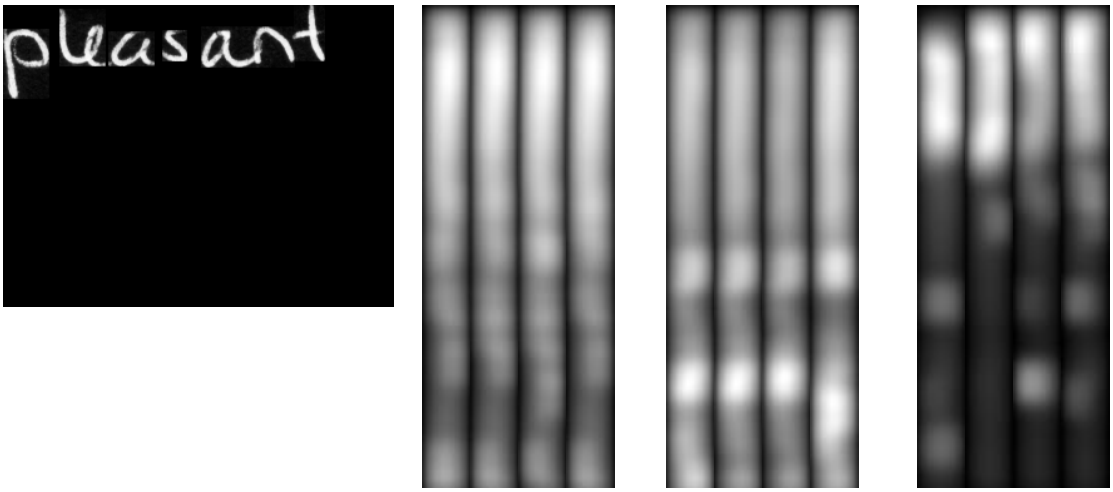

Figure 5.19 Attention Map pleasant – (Left) Image of pleasant, (Middle Left) Attention map of NN one, (Middle Right) Attention Map of NN two, (Right) Attention Map of NN three [80]

The third NN on the other hand could learn to focus on the part of the image, where the characters were positioned. This shows that an initialization different from zero is necessary, since this is the only difference between the second and third NN. Another example shows that the third NN could learn to direct its attention on useful positions in the word "Farlingham" in Figure 5.20. The characters in the image of this word are centered at different heights, making it necessary that the NN not only attends to the upper part of the image, but also to attend to a lower height for the character "g". The NN is

Figure 5.20 Farlingham Example [80]

able to pay more attention to the upper part of the image (Figure 5.21) when the character "n" is the major part in the window, even when some parts of the character "g" are present in the window. It spreads its attention and shifts it down only, when the character "g" is

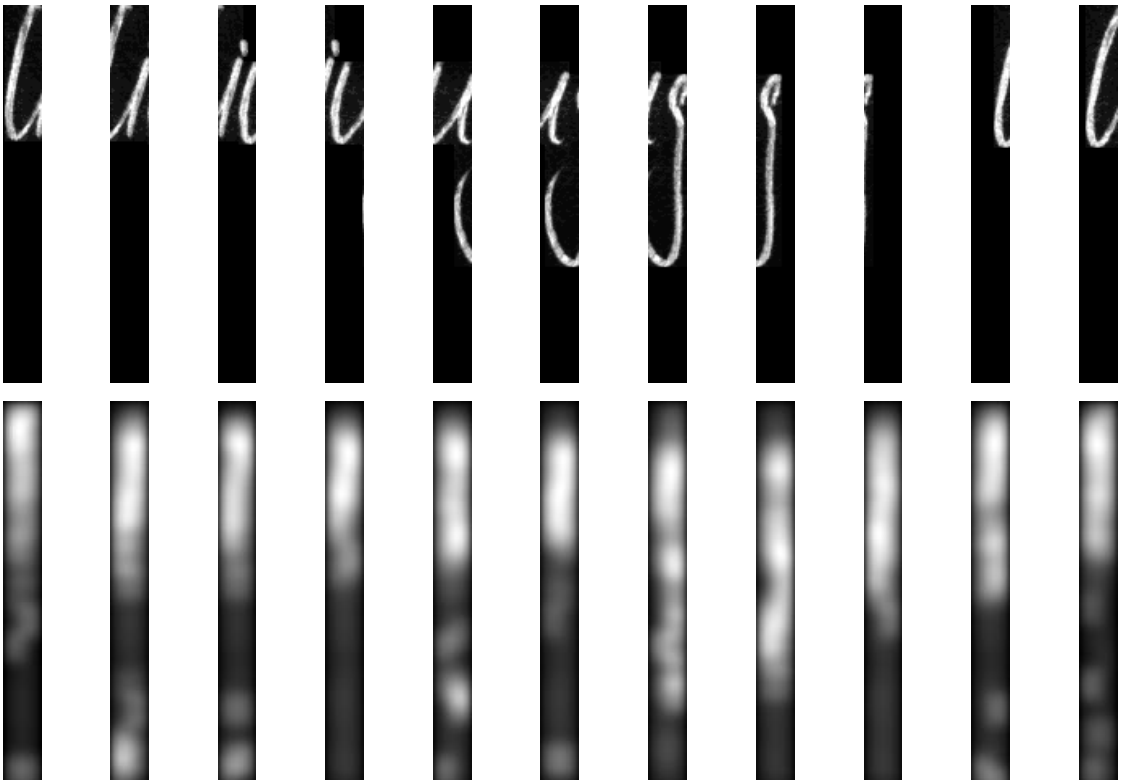

Figure 5.21 Farlingham Attention Map – (Up) Example Windows, (Bottom) Corresponding Attention Map [80]

focused more by the window. At the beginning of the character "h" it shifts again its attention, but this time upwards and it also concentrates it more. While this shows that the third NN has learned to make use of its attention to focus on some positions in the image, this does not seem to have much influence on the results, because the third NN has only a higher CER than the second NN. The first and second NN didn't seem to have

Figure 5.22 Images of Problematic Words – (Left) issue, (Middle) was, (Right) in [80]

learned making use of its attention, but instead spread it rather evenly over the whole image. The reason why the attention is not so necessary to perform well, might be correlated to the resulting sizes of the images after the CNNs, which are of a width of 3 and a height of 31. Only the height has a meaningful spatial extension, while the width is already very small, so that no real attention over those parts might be necessary or even be beneficial. It had the tendency to always focus the center width of the image. While it did learn to focus on the important parts in the image, this might not be as important, because the biggest part of the images is padding, and is therefore zero from the beginning. Many characters are of the same height or don't derive much from an average height of the characters in the image. One way to analyze the influence of attention on a window base approach better, would be the use of wider windows, so that more characters would be present in one window, but keeping the stride of each window relatively small. So, it only needs to focus in one character at every time step. The window based approach is somehow similar to hard attention. [85] [86], because it allows the NN to select small windows, which are then fed to it. The major difference is, that in the window based approach the programmer decides, which part is fed to the NN and not the NN itself, but it still sees only a small part of the image at each time step in both cases. All three of them make errors, when recognizing different crossed over characters, which is probably for the same reason as in the case of the sliding window approach. Some errors, that occur with this approach, are probably caused by the way the characters are written, examples are the words "was", "issue" and "in" in Figure 5.22. Those words are partially hard to read even for humans, because some of the characters are not written very clearly. Other mistakes occur, when the windows of the first character allow a different interpretation of the character, as is the case in the windows of the word "drunk" (Figure 5.23). The NN returns the sequence "omak", where at least the beginning seems to be somewhat reasonable, because the first four windows can be interpreted as representing the character "o". This again hints at the fact, that more global information or information from more windows is necessary, which is only received at a later step.

Figure 5.23 Split up Character – (Left) The Image of drunk, (Right) Windows of the character d in drunk [80]

5.2.4. Comparison of the different Architectures

While all three different Neural Network architectures are able to recognize handwritten characters more or less, there is a clear difference between the mistakes they make and the resulting accuracy. A comparison of them with each other and some state of the art results is therefore necessary, to evaluate them in a more general setting. The results of each best model for each architecture are listed in Table 5.6

Encoder Decoder NN	Sliding Window NN	Sliding Window Attention NN	Bluche et. Al. [53]
29.2%	11.4%	13.1%	12.6%

Table 5.6 Comparison of the different architectures

In addition to the results of the three different NN architecture the results of are [53] listed, which serves as comparison to other approaches. Only words, that made solely use of deep learning, where taken into consideration for comparison. The approach in [53] used a combination of Multi-Dimensional Long Short-Term layers [87], convolutions, a hybrid soft attention mechanism and LSTMs to classify handwritten characters in images of words, sentences and whole paragraphs from the IAM dataset. In the work by Bluche et. Al. an accuracy of 12.6% on one word where reached, while the Sliding Window NN reached an accuracy of 11.4%, the Sliding Window Attention NN an accuracy of 13.1% and the Encoder Decoder NN an accuracy of 29.2%. The Sliding Window NN reached the best results by 1.2% on the one word IAM challenge. However, an important fact needs to be made clear: even though the accuracy on single words in the IAM set was reported in the work by Bluche et. Al., the main goal was the handwritten recognition on whole paragraphs without explicit line segmentation. The NN archived an increasing accuracy for a bigger number of words that needed to be recognized at the same time.

The best results where archived for full lines with an CER of 7.0%, which is better than the Sliding Window Neural Network by 4.4%. This hints at the fact, that attention is more beneficial, when the sequence length grows. All three implemented NNs archived different results, but the Sliding Window NN and the Sliding Window Attention NN both show similarities in the types of errors they make and archive also a similar accuracy on the test set. While the Encoder Decoder NN learned more a language model, than recognizing the handwritten characters contained in an image, the other Sliding Window NN types learned to recognize the content in images instead of a language model. When the Encoder Decoder NN made a mistake, it tended to output real words, but it did not depend that strong on the input image, except the length. The Sliding Window NNs tended to make errors, which were more related to the content of the image, for example recognizing multiple characters in the windows of one character. A major reason for the similarity between the Sliding Window NN and the Sliding Window Attention NN is probably the fact, that both NNs have a very similar architecture with the attention being the major difference. The Encoder Decoder NN has the disadvantage, that all the image information must be stored in the state, which is not the case for the other two NNs. Those NNs receive parts of the image as input at each time step, allowing it to store more relevant information in its state. The Encoder Decoder NN had the advantage, that it received more global information of the image and not only local information at every time step. The reason why the Sliding Window Neural Networks did not so much learn a language model, might be the fact, that they never received information about the last interpreted character. The Encoder Decoder NN has clear information about the sequence until the current time step, since the correct label of the last time step is passed to it. This might also be influenced by the fact, that the Sliding Window NNs had multiple possibilities to generate the same output sequence, because the CTC loss was used. The Sliding Window NNs behaved more consistently, when used in training or inference, while the Encoder Decoder NN was very inconsistent between both cases, because of the changed conditions at training and testing. The main difference for the Sliding Window nets, that occurs between training and inference, is the fact, that at training time, the probability of the correct sequence after the transformation is maximized over all possible paths, while at inference the most probable sequence was either approximated by taking the most probable characters after the Softmax or it is approximated using Beam search. However, this does not influence the behavior of the RNN, because it never receives any feedback from the NN output. Another major difference between the two Sliding Window NN and the Encoder Decoder NN is, that the Encoder Decoder NN can incorporate global

information from the beginning. An example, is the number of characters in the word. The Sliding Window NNs can only infer the local windows, while the Encoder Decoder NN learns to estimate the length of the word from the total image. The reason for the difference between the two different Sliding Window NNs is not that clear, since the attention model could have learned, at least in principle, to put equal weight on each part of the input image. Therefore, receiving the same input as the Sliding Window NN, multiplied by a constant factor, which could also be prevented by adapting, increasing the parameters in the RNN accordingly. The first two attention models without initialization seemed to have learned to create an attention map, which puts a more equal weight on each part of the image. Though those have an even lower accuracy, than the version that learned to direct its attention in a more meaningful way. The reason, why the attention mechanism decreases the performance, might be caused by the reason, that the input windows are only a small part of the input image, making an additional soft attention mechanism unnecessary. Attention might then have a bad effect on the results of the training, since some information, which can be stored in its state, might mainly be used for the attention, because the state allows a direct influence by the RNN on the attention map. To understand this behavior better, additional experiments with bigger windows sizes would be beneficial.

5.2.5. Deep Learning at the Opta Data Gruppe

Since this work was created in corporation with the Opta Data Gruppe, some practical applications of deep learning at a corporation were also part of this work. One of them is the application of deep learning to recognize handwritten digits from scanned images of so called “Kassentrennblätter” (Figure 5.24). The goal was, to recognize the seven digits, from zero to nine contained in the squares, in the correct order. While one approach would have been, to extract the digits separately by extracting the squares first, this was not necessarily practical for multiple reasons. The first reason is, that the position of the seven squares vary among the images, because the scanned image of the document was not always perfect, since it could have been translated or rotated. Another problem were the numbers written in the squares. While most of them were written clearly into the squares, some of them were written close to edges of the square or crossed the edge of one square and entered another square. The alternative solution was to cut out a square of the image that contained all seven squares. This approach also needed account for the possible translations and rotations, which was done by making the square bigger, so that the squares of the transformed images still were a part of the cut-out square. Another goal

was two show some capabilities of deep learning in an applied setting. Therefore, the more complex task of extracting all seven characters from one image was chosen. To solve this problem, a Neural Network consisting of an CNN and a RNN was developed and implemented using the framework, which was part of this work. The architecture of the NN is essentially the Encoder Decoder NN. The purpose of the CNN was again to encode the image in a meaningful way, which is then used to initialize the state of the RNN. The RNN is then used to extract the necessary information and to return the correct sequence, while the last returned character, at inference, or the last correct character is passed to it at each time step. The CNN consist of three repetitions of the same structure, which were used in the CNN before: a convolution with filter size followed by a ReLU activation function, and a MaxPooling layer. The RNN consisted of one layer with 512 GRU cells. As optimization procedure, the Adam optimizer was used with a learning rate of the remaining parameters, as recommended in the paper [18]. The square, that was cut out from the input image, started at the pixel 44, 448 and had a width of 615 and a height of 136 pixels. All parameters of the NN are listed in Table 5.7. The Dataset, that was used, contained 1104804 labeled examples, which were split up into a training and a test set. The training set contained 110000 randomly chosen examples, the test set the remaining 4804 examples. A batch size of 15 was chosen for training. Since the Encoder Decoder approach was used and all sequences had the same length, the RNN runs always for seven time steps. The training took roughly 3.5 hours. The NN reached a performance of 91.6% on the test set, but it showed a similar behavior to the Encoder Decoder NN applied on the IAM dataset. When the last correct label was used as input to the RNN instead of the last output of the RNN, an accuracy of 93.6% could be archived. Even though there is a difference between both situations. The difference is not as big as it was for the Encoder Decoder NN, which was trained on the IAM, where the difference was 9.3%. There could be multiple different reasons for this, the most obvious being the fact that the RNN now has 512 GRU cells and not 256, as it was the case before. This could allow to store more information, therefore allowing to rely more on the information stored in its state. Another reason might be, that the number of time steps is now fixed, instead of being different for

Figure 5.24 Kassentrennblatt Example

Num. Conv, Layers:	Num. RNN Layers:	Num. RNN Cells:	Conv. Size 1:	Conv. Size 2:	Conv. Size 3:	Adam alpha:	Adam beta1:	Adam beta2:	Adam epsilon:
3	1	512	5×5 ×1 ×16	5×5 ×16 ×32	5×5 ×32 ×64	10^{-4}	0.9	0.999	10^{-8}

Table 5.7 Parameters for the "Kassentrennblatt" NN

each example. This might influence, what and how the RNN stores its information, because information like the number of characters in the word are not necessary anymore. Furthermore, different numbers of elements might have different requirements on how to store those information's. Therefore, making it hard for the RNN to adapt to multiple different sequence lengths at the same time. The number of different classes is now smaller, too. This makes it possible, that less information needs to be stored in its state. It is only necessary to differentiate between 10 different input characters in the image and not 78, as was the case before. Another reason might be, that the data was not structured, so that it would be beneficial to learn a model of it. Before it learned to create a model of English words, which was possible, because of the set of different words, especially for short words, was rather restricted and contained some form of structure. This might not be the case for the given data set, since to many different combinations of numbers are relatively equal likely. Even though not all combinations of seven digits were contained in the test set. This should show that Deep Learning could be useful for the company and it indeed archived this aim. Other applications of deep learning in the company were discussed and an additional one was tested. The task was, to classify different types of forms into two different classes. Using a CNN similar to before, the task was performed, but this time with four repetitions of the structure, reaching an accuracy of roughly 98.6%

6. Conclusion and Further Work

6.1. Conclusion

During this work, an OpenCL deep learning framework for handwritten character recognition as well as different Neural Network architectures for perform handwritten character recognition were successfully implemented. The framework supported the fast creation of different Neural Networks including CNNs, RNNs and combinations of both. It was successfully tested on all three them where it took roughly 3.5 hours. It was build

modular and easily extensible to be able to include future developments in the field of deep learning. The OpenCL API was encapsulated by the backend to allow an easy change of the underlying heterogenous programming system, for example to change it to CUDA. The training of the Neural Networks is performed using OpenCL, thereby achieving a cross-platform compatibility. The platforms still need to support C++ 11. Even though it is flexible enough to allow the creation of different architectures, the overhead is kept relatively low by creating everything necessary at initialization. This initialization and the resulting static unrolling of the graph creates a disadvantage, when the number of time steps an RNN must run varies. Two different convolution OpenCL kernels were implemented where the OpenCL kernel based on [66] does not achieve the expected results so that further optimization is necessary. The GPU memory used by the NN system is kept low by implementing only kernel functions which require no additional global memory and reducing the number of temporary data buffers to a minimum. This allows the training of deeper models or the use of bigger batches. Since the handling of data is also an important part of deep learning, an asynchronous data loader was successfully implemented. The loader allows the easy adaption to different data-sets. The framework was successfully applied to train models for recognizing multiple handwritten numbers contained in an image.

To perform handwritten character recognition, three different Neural Networks architectures were trained using the IAM data-set. The first one was used as baseline and is based on an Encoder Decoder pattern, it reached an accuracy of 29.2%. The low accuracy is mainly based on the face that it learned a word model biased by the input image. It did only learn to recognize specific characters in the input in a way restricted way. The second and third model are based on a sliding window approach, meaning that the input image is split up into multiple possible overlapping images. The first sliding window model reached an accuracy of 11.4%. It archived results comparable to the literature on the same tasks but it is a little bit worse, by 4.4%, compared to a model trained on full sentences. It has shown that bigger windows with some overlap archive better results, with a difference of 8.5%, compared to smaller windows that don't overlap. Additionally, it showed that the size of the first RNN layer has the biggest influence, with a difference of 5.9%, on the resulting accuracy. Both sliding window models encountered the biggest problems, when it was necessary to recognize global features due to the local nature on how the input is passed to the Neural Networks. The last implemented Neural Network architecture expanded upon the sliding window model by incorporating an attention mechanism. It archived an accuracy of 13,1%. It was necessary to initialize the

state of the first RNN layer using a transformation of the input window. Only this way the Neural Network learned to focus on the important parts using the attention mechanism. If zero initialization was used the attention mechanism was not really used to focus on something specific.

In the context of this work Deep Learning was established to the Opta Data Group in Germany. Different fields of applications of deep learning in the company were explored and it was introduced to some employees of the company. It was successfully applied to two different tasks related to character recognition. The tasks were the recognition of seven handwritten numbers and the classification of images of form into two classes. The achieved accuracy for the number recognition is 91.6% and 98.6% for the classification of forms. An incorporation into the production must still be done and requires more work.

6.2. Further Work

While a framework for applying deep learning to handwritten character recognition and other deep learning applications was developed in this work, there can still much be done to improve it. The framework has a few points, where it is rather static in the way, it can be employed. One example is the number of time steps, that get executed, when the NN contains a recurrent part. It is necessary, that the NN always runs for the same number of time steps, so the NN is unrolled statically. This can be computationally expensive, because the results of many computations are not used. Moreover, the padding must always be adapted to the longest size, so that the RNN always accesses data, which is still in the range of the buffers. The same holds true for the batch size and the size of the input data. The batch size must be fully specified, when the NN is initialized. This might not be a problem for training, because at training time the batch size stays in general the same, but at inference time this might prove to be a problem. An example for a situation would be a system for handwritten character recognition, which is used in a cloud and accessed by many users. To utilize the given hardware better, it might be useful to run the NN on many images at the same time, but on the other hand, there might not always be enough images to fill such a batch. It might be also not okay, to wait long until enough images are available, since this could take potentially much time. The system would need to be flexible enough, to allow the use of different batch sizes in different situations. A possible solution to these problems would be, to implement a variable system, which allows to infer the correct size for each variable at runtime. This way the size of the input image, the length of the sequence and the batch size could be inferred at runtime. This way the framework would obtain much more flexibility, but care must be taken, that the

performance of the system does not degrade too much. Ideally it would be possible to allow the use of constant variables, which marks the variable as static, so that no runtime inference is necessary. One possible way to implement such a system would be, to create an acyclic graph, where the variables are edges and operations are nodes. The nodes would then calculate, how the input size is transformed to another size, which is already done in the current state of the framework, but only at the initialization. This system would require a few changes to the current framework, since the OpenCL buffer objects must be created and managed in a more flexible way, while they are rather static now. Adaptions with respect to the loop unrolling would be necessary, to allow dynamical sequence sizes for RNNs. Another part of the framework, that could be improved, would be the kernel compilation, since the size of the work items and the parameters of the kernel objects must be specified by the user now, or a default configuration must be used. It would be useful, to implement an automatic optimization of the operations required for the NN at initialization time. This way, the operations could adjust itself especially to new devices, making it more flexible to use and making less knowledge of the specific hardware necessary, to use more optimal. It could be done by iteratively optimizing the kernel code [88]], or alternatively by using machine learning, to train a model, which then learns to return the probable runtime for different configurations, allowing a direct comparison for different configurations without the need to run them [89]. It would as well be useful, to be able, to merge multiple kernels at runtime, which are executed one after each other. This could reduce the number of calls to OpenCL run kernel functions, which might be quite computationally expensive. Further optimization of the kernel functions could improve the performance of the Framework as well.

The best Neural Networks used in this work, archived results, which are comparable to [53], nevertheless this work had another focus. There is still space for improvement of the NN, especially the Sliding Window NNs offer many ways to advance. The CNN used in this work, was kept the way like for almost all model One disadvantage of ResNet among others is, that it needs relatively much memory, since the number of parameters in the model is relatively big. This is not necessarily a problem, when GPUs are used, but this might become a problem, when other hardware devices with more restricted memory are used. Examples are FPGAs and embedded systems, which have in many cases not so much on chip memory available. Therefore, big models might be a problem. Lower memory requirements would make it more practical to update a NN on a system more often. A solution to this is the SqueezeNet architecture [75], which achieves a performance comparable to that of the AlexNet , but needs less than one MB of memory.

They achieved this by incorporating two different layer types, the squeeze layers and the expand.

Another way to reduce the size of the model and to possibly reduce the execution time would be, to use a fixed numerical precision [90], or a reduced numerical precision [91]. All of this can help to improve, to increase the memory efficiency on different devices and allow them to be used in a more flexible way. At the same time, the accuracy can be increased by using bigger CNNs. There are also other ways, that can possibly increase the accuracy of the NN, which aim at improving the RNN part of the model. One part of the NN, for which more exploration is needed, is the accuracy in dependence of the window width and offset, since a change of both at the same time increased the accuracy drastically. An interesting question would be, whether the accuracy could be improved by changing the size of the window and the offset of the window. This could be especially interesting for the attention version, since it only learned to direct its attention in the height dimension of the image, but not in the width. This could be due to the small window width after the convolutions. Attention models, which are trained with bigger window sizes, could therefore achieve a better accuracy, since more information is available. This gives the system more choices on which part it deems important at which step. Futhermore, the bigger window size could help the system to recognize characters, which are wider, since those characters were sometimes a problem for both sliding window NNs architectures. The attention mechanism itself could be improved as well. One way would be to use a small CNN, to calculate the attention weights, instead of using a normal feed forward NN, which only contains matrix multiplications. CNNs perform, in general, better on images as normal feed forward Neural Network. Thus, it could be worth exploring. To incorporate the state of the RNN into the attention mechanism, when an CNN is used, some part of the Attention Neural Network should still contain a matrix multiplication. Since the state is no image, no invariance like translation invariance can be assumed. All Neural Networks in this work, except for one, used a zero initialization of the state, but this could also be done differently. The normal sliding window Neural Network could be initialized in the same way, the last attention NN was initialized. In this case, not only the first layer is initialized this way, but all of them using custom parameter matrix. In the attention NN, all layers could be initialized this way as well. Instead of using only the CNN encoding of the first window to initialize the NN, the full image in a down scaled version could be used allowing, the NN to receive global information from the beginning. This could help to improve the performance of the sliding window NNs, since both versions seemed to have problems, when more global information where

required or beneficial. An example for this problem is the fact, that all NNs had specific problems with the crossed over words, which were not easy to recognize, when only the first few windows were available. While this is one way to make global information more accessible to the NN, another approach would be, to use bidirectional RNNs [92]. This expands the normal RNN by an additional RNN, which gets the sequence in opposite order from the end to the start. This bidirectional RNN would then create an encoding for the current window consisting of information from the start window to this window and from the end window to this window by concatenating them. An additional RNN would then take this output for each window and creates an output sequence. The additional RNN would receive more global information, because the input is based on all windows from the beginning and the end to this point. This could alleviate the problems; the current sliding window architectures have with global information and thus increase the accuracy.

7. Bibliography

[1] S. Raschka, „Single-Layer Neural Networks and Gradient Descent," 24 Mar. 2015. [Online]. Available: http://sebastianraschka.com/images/blog/2015/singlelayer_neural_networks_files/perceptron_schematic.png.

[2] F. Rosenblatt, „The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review,* Nr. 65, pp. 386-408, Nov. 1958.

[3] M. Hornik, M. Stinchcombe und H. White, „Multilayer feedforward networks are universal approximators," *Neural Networks,* Nr. 2, pp. 359-366, 1989.

[4] R. Hahnloser, R. Sarpeshkar, M. Mahowald, R. Douglas und H. Seung, „Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature,* Nr. 405, pp. 947-951, 2000.

[5] I. Arel, D. Rose und T. Karnowski, „Deep machine learning-A new frontier in artificial intelligence research," *Computational Intelligence Magazine,* Nr. 5, pp. 13-18, 2010.

[6] D. Petrova und A. Solev'ev, „ The Orgin of the Method of," *Historia Mathematica,* Nr. 24, pp. 361-375, 1997.

[7] V. Vapnik, „An overview of statistical learning theory," *IEEE Transaction on Neural Networks,* Nr. 10, pp. 988-999, 1999.

[8] H. Robbins und S. Monro, „A stochastic Approximation Method," *The Annals of Mathematical Statistics,* Nr. 22, pp. 400-407, 1951.

[9] R. Byrd, G. Chin, J. Nocedal und Y. Wu, „Sample size selection in optimization methods for machine learning," *Mathematical programming,* Nr. 134, pp. 127-155, 2012.

[10] A. Shapiro und Y. Wardi, „Convergence analysis of gradient descent stochastic algorithms," *Journal of Optimization Theory and Applications,* Nr. 91, pp. 439-454, 1996.

[11] O. Dekel, R. Gilad-Bachrach, O. Shamir und L. Xiao, „Optimal Distributed Online Prediction Using Mini-Batches," *Journal of Machine Learning Research,* Nr. 13, pp. 165-202, 2012.

[12] R. Sutton, „Two problems with backpropagation and other steepest-descent learning procedures for networks," in *Proc. of 8. Annual Conference of the Cognitive Science Society,* 1986.

[13] D. Rumelhart, G. Hinton und R. Williams, „Learning internal representations by error propagation," *Parallel Distributed Processing,* Nr. 1, pp. 318-362, 1987.

[14] Y. Nesterov, „A method of solving a convex programming problem with convergence rate O(1/sqr(k))," *Soviet Mathematics Doklady,* Nr. 27, pp. 372-376, 1983.

[15] J. Duchi, E. Hazan und Y. Singer, „Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research,* Nr. 12, pp. 2121-2159, 2011.

[16] M. Zeiler, „Adadelta: An adaptive learning rate method," *arxiv.org:1212.5701,* 2012.

[17] G. Hinton, „Coursera,“ [Online]. Available: https://www.coursera.org/learn/neural-networks/lecture/YQHki/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude.

[18] D. Kingma und J. Ba, „ADAM: A Method for Stochastic Optimization,“ *arXiv:1412.6980,* Dec. 2014.

[19] S. Ruder, „An overview of gradient descent optimization algorithms,“ *arXiv:1609.04747,* 2016.

[20] D. Rumelhart, G. Hinton und R. Williams, „Learning representations by back-propagating errors,“ *Nature,* Nr. 323, pp. 533-536, 1986.

[21] Stanford University, „Convolutional Neural Networks (CNNs / ConvNets),“ 2017. [Online]. Available: http://cs231n.github.io/assets/nn1/neural_net2.jpeg.

[22] J. Hochreiter, „Untersuchungen zu dynamischen neuronalen Netzen,“ 15 Juni 1991.

[23] S. Hochreiter, Y. Bengio, P. Frasconi und J. Schmidhuber, „Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,“ *A Field Guide to Dynamical Reccurent Neural Networks.*

[24] X. Glorot, A. Bordes und Y. Bengio, „Deep sparse rectifier neural networks,“ *Proc. of 14th International Conference on Artifical Intelligence and Statistics,* pp. 315-323, 2011.

[25] J. van Doorn, „Analysis of deep convolutional neural network architectures,“ 2014. [Online]. Available: http://referaat.cs.utwente.nl/conference/21/paper/7438/analysis-of-deep-convolutional-neural-network-architectures.pdf.

[26] Y. LeCun, B. Boser, J. Denker, D. Henderson, D. Howard, W. Hubbard und L. Jackel, „Backpropagation applied to handwritten zip code recognition,“ *Neural computation,* pp. 541-551, 1989.

[27] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, „Gradient-based learning applied to document recognition,“ *Proc. of 86th IEEE,* pp. 2278-2324, 1998.

[28] Stanford University, „Convolutional Neural Networks (CNNs / ConvNets),“ 2017. [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[29] K. He, X. Zhang, S. Ren und J. Sun, „Deep residual learning for image recognition,“ *arXiv:1512.03385,* 2016.

[30] A. Colyer , 22 Mar. 2017. [Online]. Available: https://adriancolyer.files.wordpress.com/2017/03/rethinking-inception-fig-1.jpeg?w=480.

[31] K. Simonyan und A. Zisserman, „Very deep convolutional networks for large-scale image recognition,“ *arXiv:1409.1556,* 2014.

[32] C. Olah, „Understanding LSTM Networks,“ 27 Aug. 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-rolled.png.

[33] M. Jordan, „Serial order: A parallel distributed processing approach,“ *Advances in psychology,* Nr. 121, pp. 471-495, 1986.

[34] J. Elman, „Finding structure in time,“ *Cognitive science,* Nr. 14, pp. 179-211, 1990.

[35] C. Olah, „Understanding LSTM Networks,“ 27 Aug. 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-unrolled.png.

[36] M. Mozer, „A focused back-propagation algorithm for temporal pattern recognition," *Complex systems,* Nr. 3, pp. 349-381, 1989.

[37] P. Werbos, „Backpropagation through time: what it does and how to," *Proc. of 78th IEEE,* pp. 1550-1560, 1990.

[38] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot und A. Graves, „Memory-efficient backpropagation through time," *arXiv:1606.03401,* 2016.

[39] S. Hochreiter und J. Schmidhuber, „Long short-term memory," *Neural Computation,* Bd. 8, Nr. 9, pp. 1735-1780, 1997.

[40] R. Pascanu, T. Mikolov und Y. Bengio, „On the difficulty of training recurrent neural networks," *Proc. of 30th International Converence on Machine Learning (ICML),* 2013.

[41] K. Cho, B. van Merrienboer, D. Bahdanau und Y. Bengio, „On the properties of neural machine translation: Encoder-decoder approaches," *arXiv:1409.1259,* 2014.

[42] J. Chung, C. Gulcehre, K. Cho und Y. Bengio, „Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv:1412.3555,* 2014.

[43] A. Graves, S. Fernandez, F. J. Gomez und J. Schmidhuber, „Connectionist temporal classification: Labeling unsegmented sequence data with recurrent neural nets," *Proc. of 23rd Interantional Conference on Machine Learning,* pp. 369-376, 2006.

[44] L. R. Rabiner, „A tutorial on hidden markov models and selected applications in speech recognition," *Proc. IEEE,* pp. 257-286, 1989.

[45] A. Graves und J. Schmidhuber, „Offline handwriting recognition with multidimensional recurrent neural networks," *Advances in Neural Information Processing Systems,* Nr. 21, pp. 545-552, 2008.

[46] K. Cho, B. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk und Y. Bengio, „Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv:1406.1078,* 2014.

[47] O. Vinyals, A. Toshev, S. Bengio und D. Erhan, „Show and tell: a neural image caption generator," *arXiv:1411.4555,* 2014.

[48] O. Vinyals, A. Toshev, S. Bengio und D. Erhan, „Show and tell: a neural image caption generator," *arXiv:1411.4555,* 2014.

[49] D. Bahdanau, K. Cho und Y. Bengio, „Neural machine translation by jointly learning to align and translate," *arXiv:1409.0473,* 2014.

[50] D. Bahdanau, K. Cho und Y. Bengio, „Neural machine translation by jointly learning to align and translate," *arXiv:1409.0473,* 2014.

[51] A. Graves, „Sequence transduction with recurrent neural networks," *In Proc. of 29th International Conference on Machine Learning (ICML),* 2012.

[52] K. Xu, J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel und Y. Bengio, „Show, attend and tell: Neural image caption generation with visual attention," *arXiv:1502.03044,* 2015.

[53] T. Bluche und R. Messina, „Scan, attend and read: End-to-end handwritten paragraph recognition with mdlstm attention," *arXiv:1604.03286,* 2016.

[54] Khronos Group, „OpenCL," Khronos, [Online]. Available: https://www.khronos.org/opencl/.

[55] Khronos OpenCL Working Group, „The OpenCL Specification Version 1.2, Revision 19," 14 Nov. 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[56] „OpenCL," Intel, [Online]. Available: https://software.intel.com/en-us/intel-opencl.

[57] „OpenCL," NVIDIA, [Online]. Available: https://developer.nvidia.com/opencl.

[58] „OpenCL," AMD, [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/.

[59] „OpenCL," Xilinx, [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[60] A. Munshi, „The OpenCL Specification," Khronos Group, 2012.

[61] NVIDIA, „NVIDIA OpenCL best practices guide," 10 Aug. 2009.

[62] E. Bendersky, „Variadic templates in C++," 24 Oct. 2014. [Online]. Available: http://eli.thegreenplace.net/2014/variadic-templates-in-c/.

[63] ISO/IEC, „Programming Language C++.," ISO International Standard ISO/IEC 14882:2011, 2014. [Online]. Available: https://www.iso.org/standard/50372.html.

[64] G. Tan, L. Li, S. Treichler, E. Phillips, Y. Bao und N. Sun, „Fast implementation of DGEMM on Fermi GPU," *Supercomputing 2011*, Bd. 35, pp. 1-11, 2011.

[65] K. Chellapilla, S. Puri und P. Simard, „High performance convolutional neural networks for document processing," in *10th Workshop on Frontiers in Handwriting Recognition*, 2006.

[66] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro und E. Shelhamer, „cuDNN: Efficient primitives for deep learning," *arXiv:1410.0759*, 2014.

[67] Y. Jia, „Caffe: An open source convolutional architecture for fast feature embedding," 2013. [Online]. Available: http://caffe.berkeleyvision.org/.

[68] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley und Y. Bengio, „Theano: a CPU and GPU math expression compiler," in *Proc. of Python for Scientific Computing Conference (SciPy)*, 2010.

[69] R. Collobert, K. Kavukcuoglu und C. Farabet, „Torch7: A matlab-like enviornment for machine learning," *BigLearn, NIPS Workshop*, 2011.

[70] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu und X. Zheng, „Tensorflow: Large-scale machine learning on heterogenous systems," *arXiv:1603.04467*, 2015.

[71] F. Tschopp, „OpenCL Caffe," [Online]. Available: https://github.com/BVLC/caffe/tree/opencl.

[72] „Torch7: Cheatsheet," 2017. [Online]. Available: https://github.com/torch/torch7/wiki/Cheatsheet.

[73] LISA lab., „OpenCL Theano," 2017. [Online]. Available: http://deeplearning.net/software/theano/tutorial/using_gpu.html.

[74] M. Mathieu, M. Henaff und Y. LeCun, „Fast training of convolutional networks through ffts," *arXiv:1312.5851*, 2013.

[75] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally und K. Keutzer, „Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5mb model size," *arXiv:1602.07360,* 2016.

[76] T. Wang, D. Wu, A. Coates und A. Ng, „End-to-end text recognition with convolutional neural networks," *Proc. of 21st International Conference on Pattern Recognition*, pp. 3304-3308, 2012.

[77] M. Jaderberg, K. Simonyan, A. Vedaldi und A. Zisserman, „Reading text in the wild with convolutional neural networks," *arXiv:1412.1842,* 2014.

[78] M. Sabatelli, Y. Shkarupa und R. Mencis, „Offline Handwriting Recognition Using LSTM Reccurent Neural Networks," 2016.

[79] C. Nugteren, „Tutorial: OpenCL SGEMM tuning for Kepler - Kernel 6: 2D register blocking," 2014. [Online]. Available: https://cnugteren.github.io/tutorial/pages/page8.html.

[80] U. Marti und H. Bunke, „The IAM-database: An english sentence database for off-line handwriting recognition," *Journal on Document Analysis and Recognition,* Nr. 5, pp. 39-46, 2002.

[81] Tensorflow, „Sequence-to-Sequence Models," Google Brain, 2017. [Online]. Available: https://www.tensorflow.org/tutorials/seq2seq.

[82] „Tensorflow API," 26 Apr. 2017. [Online]. Available: https://www.tensorflow.org/api_docs/python/.

[83] NVIDIA, „CuDNN," [Online]. Available: https://developer.nvidia.com/cudnn.

[84] CMSoft, „Case study: High performance convolution using OpenCL __local memory," 2015. [Online]. Available: http://www.cmsoft.com.br/opencl-tutorial/case-study-high-performance-convolution-using-opencl-__local-memory/. [Zugriff am 21 06 2017].

[85] J. L. Ba, V. Mnih und K. Kavukcuoglu, „Multiple object recognition with visual attention," *arXiv:1412.7755,* 2014.

[86] K. Xu, J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel und Y. Bengio, „Show, attend and tell: Neural image caption generation with visual attention," *arXiv:1502.03044,* 2015.

[87] A. Graves und J. Schmidhuber, „Offline handwriting recognition with multidimensional reccurent neural networks," *Advances in Neural Information Processing Systems(NIPS),* Nr. 21, pp. 545-552, 2009.

[88] J. F. Fabeiro, D. Andrade und B. B. Fraguela, „OCLoptimizer: an iterative optimization tool for OpenCL," *Proc. of Intl. Conf. on Computational Science (ICCS),* pp. 1322-1331, 2013.

[89] T. L. Falch und A. C. Elster, „Machine learning based auto-tuning for enhanced OpenCL performance portability," *Proc. of Parallel and Distributed Processing Symposium Workshop (IPDPSW),* p. 1231–1240, 2015.

[90] V. Vanhoucke, A. Senior und M. Z. Mao, „Improving the speed of neural networks on cpus," *Deep Learning and Unsupervised Feature Learning Workshop, NIPS,* 2011.

[91] S. Gupta, A. Agrawal, K. Gopalakrishnan und P. Narayanan, „Deep learning with limited numerical precision,“ *arXiv:1502.02551*, 2015.

[92] M. Schuster und K. K. Paliwal, „Bidirectional recurrent neural networks,“ *Signal Processing, IEEE Transactions,* Nr. 45, p. 2673–2681, 1997.

8. Acknowledgement

At this point I would like to thank all the people who gave me their professional and personal support during my thesis.

At first, I would like to thank Prof. Göhringer for enabling me the great chance of doing my master thesis in the field of Deep Learning and always having some helpful suggestions for me.

It was a huge enrichment for me, to get the possibility to do my thesis in cooperation with the Opta Data Group. Here I would like to thank my colleges Torsten Nordhoff and Bastian Klinken for their great support.

Furthermore, I would like to thank my supervisor, Lester Kalms for his constructive criticism and always having a good advice for me.

My special thanks goes to my friends, my family and my girlfriend for their patience, emotional backing and always supporting me.