Master Thesis

in Computational Linguistics

at the Ludwig-Maximilians-Universität München

Faculty of Languages and Literatures

Parallelization of Neural Machine Translation

submitted by
Valentin Deyringer

Supervisor:	Dr. Alexander Fraser
Advisor :	Dr. Tsuyoshi Okita
Working Period:	March 14th – August 1st 2016

Declaration

I hereby declare that this thesis is my own work, and that I have not used any sources and aids other than those stated in the thesis.

Munich, August 1st 2016

.....
Valentin Deyringer

Abstract

Neural Machine Translation is an aspiring new approach to Machine Translation using Neural Networks. Based on large parallel corpora, recent works trained translation systems which are able to model semantics of the input language well and to transform them into meaningful translations in the output language. However, the training of such models usually takes several days. To shorten the runtime of programs, generally, parallel approaches are a natural choice. In this work the asynchronous optimization algorithms HOGWILD!, ASYNCHDA and ASYNCHADAGRAD are implemented and evaluated for their performance on training a Neural Machine Translation Model. It is shown that HOGWILD! works reasonably well but sophisticated sequential algorithms still attain better convergence properties. Additionally, a preliminary experiment on the task of handwritten digit recognition shows that the tested algorithms can be applied to train Neural Networks of different structures.

Contents

1 Introduction

Machine Translation (MT) is one of the most prominent research areas in computational linguistics. As early as 1949, the idea of tackling the problem of translation with statistical and information-theoretical methods came up (Weaver, 1949). Despite being studied for a long time, there is no unequivocal solution due to the elusive nature of the issue.

The current main line of research in MT is Statistical Machine Translation (SMT) which aims towards designing algorithms that use parallel corpora of translated text to build translation models. Apart from their dependence on large amounts of translated text, these systems also rely on the adjustment of submodules handling tasks like feature extraction or word alignment of translated phrase pairs. Based on improvements of these submodules, SMT systems are achieving ever better results. Bojar et al. (2015) assemble results of state-of-the-art SMT systems.

Many novel SMT systems make use of components based on Neural Networks (NNs). In contrast to other machine learning methods, NNs are able to learn the relevant characteristics of the data independently (Bengio et al., 2013) and thus do not rely on handcrafted features which in turn requires expert knowledge and extensive study of the data basis. Backed by growing amounts of data available and increasing computational power, NNs have achieved remarkable results in different disciplines. (Goodfellow et al., 2016) NNs have also proven to perform very well in tasks related to Natural Language Processing (NLP) like automatic speech recognition (Chan et al., 2016), image caption generation (Xu et al., 2015), and a variety of other tasks (Collobert et al., 2011; Andor et al., 2016). Also their application to MT showed considerable improvements (Cho et al., 2014b; Sutskever et al., 2014).

These promising results of adopting NNs for MT and especially their capability of capturing the semantics of phrases (Cho et al., 2014b) led to the emergence of a new branch of research referred to as Neural Machine Translation (NMT). This approach addresses the problem of translation with techniques solely based on NNs. A comparably simple system has shown that an NMT system is able to reach near state-of-the-art results and even surpass a matured SMT system (Bahdanau et al., 2014).

A major drawback of NMT systems attenuating the positive findings is the long time needed to fit the translation models. Swapping the underlying matrix computations of NN training to Graphics Processing Units (GPUs) achieves great performance improvements, since GPUs are very well suited for this kind of operations. It is possible to heavily parallelize the calculation on the many cores of GPUs speeding up training of NNs significantly (Brown, 2014). There are several libraries for programming languages which offer a convenient interface for GPU programming in the context of NNs. Nowadays, almost all real world applications of bigger NN models involve computation on GPUs.

Dependent on quantity of training data and model size, which both generally have a positive effect on the resulting models quality when increased, training NMT systems reportedly still requires several days. Training times of 3 to 10 days are common (Cho et al., 2014b; Sutskever et al., 2014; Bahdanau et al., 2014). In consequence, other ways to speed up the training are desirable. Besides from using GPUs, a way to shorten training times is parallelization on a higher level. There are generally two distinct approaches to do so, namely *model parallelism* and *data parallelism*. These approaches do not restrict the application of GPUs for the underlying matrix calculations and allow working with GPU clusters, making use of the benefits mentioned above.

The method of model parallelism distributes different computations performed on the same data onto multiple processors. The results are then merged in an appropriate way by a master process which also handles communication between processors as they are

dependent on the results computed by the other processors. This technique is well suited for NNs due to their structure and is successfully implemented for the training of NMT models in Sutskever et al. (2014). However, the work in hand is not concerned with model parallel approaches.

Data parallelism pursues a different approach where the processors perform the same operation on different data. In terms of optimization of NNs, this means that the training data is divided among the processors while shared parameters of the network are updated according to a suitable schedule. Data parallel training of NNs is not a trivial task and the commonly used optimization algorithms for training NNs are inherently iterative. Nevertheless, there are approaches in a data parallel fashion that allow parallelization of NN optimization.

The main contribution of this work is the implementation of parallelizable optimization algorithms for the training of NMT models. The final results suggest that fitting NMT models in parallel has the potential to speed up the training process. As a secondary finding, a preliminary experiment shows that the same algorithms can be applied to NNs of various structures. There are, however, some restrictions when applying the algorithms for different problems. The differences between the tested algorithms, are interestingly big.

It is found that Hogwild! is suited best for parallelized training of NMT models and that the asynchronous algorithms AsynchDA and AsynchAdaGrad are not suitable for optimization of NMT models in the tested settings. A comparison with sequential optimization algorithms shows that advanced serial optimization algorithms continue to be more performant in terms of decreasing the error of NMT models.

It is to note that the results are achieved on machines with multiple CPU cores. Nevertheless, the procedures are transferable to GPU clusters easily. While it is expected that the usage of GPUs further accelerates the speedup, an evaluation thereof is left to future research. For details about the implementations for this work and availability of source codes see Section 6.1.

The results are given in terms of the development of error the models achieve on the training data set and a separate test data set throughout the training process. Measuring the actual translation performance, for example by BLEU score[1], and further qualitative evaluation of the models is out of the scope of this work.

An important property of this work is its transferability to other sequence-to-sequence learning tasks. This is due to the property of NNs not being restricted to the type of input as long as it can be represented in vector form. However, this assumption is not universally valid, as the applied algorithms make assumptions about the sparsity properties of the underlying data. Thus, the findings may not be reproducible for other kinds of data.

This thesis commences with a summarization of publications in the field of NMT in Section 2. The most prominent works are displayed, explaining the NN architectures applied for the task of MT and listing some results achieved with NMT models. This section concerned with related work continues with an overview of the other field of research important to this study, namely parallelizable optimization algorithms. Strategies proposed for asynchronous optimization which follow a strategy similar to the ones applied for this work are listed and briefly examined.

The background of NMT systems is further explored in the following section. By starting with the basics of NNs, further structures and algorithms are presented gradually building up to the encoder-decoder architecture implemented for popular NMT systems. Section 3

[1] BLEU score (Papineni et al., 2002) is a measure commonly used to evaluate translation quality.

completes with a description of the successful attention mechanism and the generation of translations with NMT models.

Section 4 presents the optimization algorithms typically used for NN training as well as some techniques enhancing the training. The first part of the section is concerned with the sequential algorithms SGD, ADAGRAD, ADADELTA, RMSPROP and ADAM which are widely applied for optimization of different types of NNs. The asynchronous variants of optimization algorithms used in this work, namely HOGWILD!, ASYNCHDA and ASYNCHADAGRAD, are also highlighted in this section.

In a preliminary study, the application of these asynchronous algorithms to optimization of a basic NN model is tested. This experiment concerned with handwritten digit recognition on the MNIST database is explored in Section 5. While setting the findings about parallel training of NMT models in context, this experiment also shows that application of the parallelized optimization methods is possible for different problems. However, it can also be seen that ASYNCHDA does not suit optimization of NNs very well.

Different sequential and asynchronous optimization algorithms presented in the preceding sections of this thesis are tested on the task of optimizing an NMT system. The obtained results are shown in Section 6. Examining these outcomes shows that only HOGWILD! attains performance comparable to elaborated sequential algorithms. In the experiment conducted for this work ASYNCHDA and ASYNCHADAGRAD achieve inferior optimization performance and thus are inadvisable for training NMT models.

After elaborating on these results in Section 7, this thesis concludes with a comprehensive list of shortcomings and possible improvements of this work as well as an outlook on future research in Section 8.

2 Related Work

The focus of this thesis is on the combination of two lines of research, Neural Machine Translation (NMT) and parallelizable optimization in the sense of fitting NN models. In this section the publications in both fields most important for this work are presented. The technical terms special to the domains of NNs and optimization are used here without further ado as they are elucidated in Section 3 and Section 4 respectively.

This section commences by outlining the emergence and development of research in the field of NMT. These explanations are ensued by a short depiction of several results obtained on English-to-French translation tasks with NMT systems. A more in-depth description of the underlying concepts is given in section 3.

Afterwards, works introducing optimization algorithms suitable for training NNs in parallel settings will be summarized briefly in this section. Section 4 gives a broader overview about optimization algorithms used for training NNs in general and asynchronous algorithms therefor in particular.

2.1 Neural Machine Translation

As mentioned in the Introduction, NNs have attained remarkable results in different areas of research among of which is also NLP. Furthermore, continuous representations as they are internally stored in the hidden layers of NNs have demonstrated great capabilities capturing linguistic regularities (Collobert and Weston, 2008). Considering these findings, the application of NNs to MT is an auspicious approach.

However, with the standard designs of NNs only fixed-size vectors can be used as input and received as output. This contradicts a major requirement of MT and other sequence-to-sequence learning problems where input and output are of variable and different length. Primary approaches using NNs for MT are indeed affected by this shortcoming.

For example, the *phrase based continuous translation models* presented in Schwenk (2012) use a NN to estimate the translation probabilities of phrase pairs with input and output phrases limited to a maximum of seven words each. The input words are transformed into one-hot vectors from which continuous word representations are generated by applying a shared weight matrix. These representations, often called word embeddings, are concatenated and fed into a feed-forward network with several hidden layers. In an eventual layer, the obtained vector gets multiplied with different weight matrices for each word to output. Applying softmax activation functions then generates probability distributions over the target vocabulary from which the resulting sequence of words is sampled. As the vocabularies are generally large, the output vocabulary is restricted to a fixed number of most frequent words in the target language. This common approach is owed to performance and memory issues. Also, phrases shorter than seven words can be inputted by padding the input sequence with zero vectors up to the required length. It is not clearly explained what criterion is used to truncate the generated target phrase if necessary. However, the maximum lengths of input and output phrases are still restricted. In an English-to-French translation task the proposed system is shown to slightly improve translation quality when used to rescore target translations generated by an SMT system or integrated into the decoder of this system.

Another system proposed by Schwenk (2012), which is not evaluated, does not generate the words from the same hidden layer but rather hierarchically samples target words by conditioning their probabilities on the hidden states of the previously generated word. This approach is similar to the *recurrent language model* presented by Mikolov et al. (2010), but still restricts the length of the output to a previously fixed number of words.

The problem of translating phrases can be extended to the sentence level, where the limitation of fixed input and output persists. In the following, only sentence based approaches are considered which is the major approach of MT systems in general. To overcome the size restrictions, NN architectures allowing variable input and output lengths are desirable. Also, Recurrent Neural Networks (RNNs) do not circumvent the problem of fixed input and output sizes, as they only generate one output vector for each part of the input sequence or one output vector for the whole input sequence. Therefore, RNNs are suitable for problems where the alignment of input and output corresponds and is known beforehand like for example in part-of-speech tagging.

Motivated by the properties attributed to continuous space representations, Kalchbrenner and Blunsom (2013) developed two translation systems. Starting from a sequence of continuous word representations resembling the source sentence, the first system condenses the source sentence into one vector using several convolutional layers. Since the number of convolutions needed to obtain just one vector is not constant, an additional weight matrix is applied in settings where no further convolutions can be applied but still more than one vector is present. Each intermediate convolution layer can be viewed as an n-gram representation of the source sentence. The target sentence is then generated word by word using a recurrent language model as presented in Mikolov et al. (2010). This language model is conditioned on the sentence encoding vector for the generation of each new word. While inducing the semantics of the source sentence at each generation step, this system does not incorporate the fact of the target words being dependent on different parts of the original sentence.

To counteract this shortcoming, a second system applies the same convolutional operations to reduce the number of input vectors such that they represent the n-grams of the source sentence with a specific value of n. In the evaluated system, n is set to 4. From the intermediate vectors representing the four-grams in the source sentence, inverse convolutional operations are applied to expand the number of vectors again. The number of vectors is increased until a previously predicted length of the target sentence is reached. The target sentence is again generated with a recurrent language model, where the generation at each step is now conditioned on the previously obtained vector at the current index. The convolution operations used to reduce the number of vectors are explained and used as a specific example in Section 3.3.

The presented systems show low perplexities[2] in comparison with other language models. In terms of BLEU score, the models used for rescoring candidate translations of a baseline SMT system perform slightly worse than the original system. These findings suggest that the recurrent language model which samples the target translation fits the target language well, while the translation capabilities are restricted.

Another way to tackle the problem of fixed size input and output is proposed by Cho et al. (2014b). This approach consists of two RNNs. One first RNN layer encodes the input into a syntactically and semantically meaningful vector representation while a second layer decodes this representation producing the target sentence one word at a time.

Due to the tasks performed by the two layers, this structure is called the *encoder-decoder* architecture. This architecture is described in detail in Section 3.5. In their work, the proposed encoder-decoder system is used to estimate the translation probabilities of phrase pairs and to rerank the candidate translations produced by an SMT system improving the results by about 0.7 points BLEU score on the test set of an English-to-French translation task. A qualitative analysis of the models trained in this work shows that the first RNN captures linguistic regularities of the input language and the semantics of the input sentences very well.

[2] Perplexity is a common measure to compare probabilistic language models, lower perplexity values are better.

The proposed system is not only able to estimate probabilities but also to sample translations without any additional components. However, this application of the proposed system is left open for succeeding work, some of which is presented hereafter.

As an additional contribution, Cho et al. (2014b) present a novel type of gated recurrent unit which is reportedly working well for the task at hand. This unit is presented more detailed in Section 3.4.2.

Cho et al. (2014a) elaborate on the properties of NMT systems focusing on the encoder-decoder model presented in Cho et al. (2014b) and a novel system utilizing a convolutional network as encoder. The used convolutional network implements a special gating mechanism which is described in Section 3.3. This mechanism reportedly induces knowledge about the grammatical structure into the encoding of the source sentence.

However, trained and tested on the same data used in Cho et al. (2014b), the two novel systems reach relatively low BLEU scores when used to translate directly. The best results are yet again achieved by using the NN model to rerank translation candidates produced by a traditional SMT system. As a major finding of this work the property of NMT systems performing worse on longer sentences is pointed out. This weakness is attributed to the fixed size vector the input sentence is condensed to failing to capture all available information.

An architecture very similar to the encoder-decoder presented in Cho et al. (2014b) is introduced in Sutskever et al. (2014). The recurrent units applied in this systems are the popular Long Short-Term Memory units (LSTMs) proposed by Hochreiter and Schmidhuber (1997). The functionality of these units is shown in Section 3.4.1. Additionally, this work uses several RNN layers, following the idea of deep learning, which is explained later. For training the system, the authors parallelize the model by distributing the computations on the different layers on several GPUs. This method corresponds to the model parallelism approach mentioned in the Introduction.

A strong point is made about the finding that reversing the order of the source sentences while maintaining the order of the target sentences improves the trained translation models significantly. Although the reason for this effect is not completely understood, the authors attribute it to newly introduced short term relationships between the beginnings of the sentence pairs. While this may hold for many language pairs like English and French for which the results were achieved on, it may not be the case for other languages where the words in the beginning of sentences do not have such matching alignment. Nevertheless, this reversion also reportedly has a positive effect on the performance of the model for long sentences which the authors attribute to a better memory usage of the deep LSTMs when reversing the input sentence. Another remark worth noting is the speedup achieved by grouping sentences of similar length in batches for estimating the gradient during training, as the variance in gradient estimates is reduced.

After training their NMT model for 10 days with 4 RNN layers for both, the encoder and decoder, it is shown that a matured SMT system is outperformed. With a combination of both systems, using the NMT model to rerank sentences produced by the SMT system achieves even better results. The ability of the encoder to capture the semantics of the source sentences is again emphasized. It is also noted that the same encoder could be used for translations into different target languages.

Despite removing the restriction of fixed input and output sequence lengths, the approaches listed so far are again limited in that they condense the input sentence into a more compact representation of fixed size. Apart from the second system in Kalchbrenner and Blunsom (2013) which reduces the number of vectors comparatively slight, the input sentence is usually merged into one vector of fixed size. The size of this vector can be inadequate to capture all information conveyed in the source sentence which especially

Publication	Task	BLEU Score
Kalchbrenner and Blunsom System I	Translation	(mean over different test sets) 21.20
Kalchbrenner and Blunsom System II	Translation	(mean over different test sets) 21.28
Cho et al.	Reranking	33.87
Sutskever et al. Best System	Translation	34.81
Sutskever et al.	Reranking	36.50
Bahdanau et al. Best System	Translation	28.45
baseline SMT system	Translation	33.30

Table 2.1: BLEU scores of different NMT systems. The outcomes were achieved on basis of different datasets. It follows that they are not directly comparable. More detailed information about the datasets is available in the corresponding publications. Reranking means the rescoring of candidate translations produced by an SMT system. The baseline SMT system is the well known system moses (Koehn et al., 2007), the corresponding BLEU score is taken from Bahdanau et al. (2014).

affects the translation of long sentences and thus becomes the bottleneck for translating long sentences (Bahdanau et al., 2014).

The so called *attention mechanism* (Graves, 2013) provides a solution to this problem. Inspired by the visual attention mechanism of humans, this approach focuses on a specific section of the input when generating an part of the output. An illustrative demonstration of the attention mechanism can be found in Xu et al. (2015) where it is applied to the task of image caption generation.

The attention mechanism is also implemented for NMT with remarkable success (Bahdanau et al., 2014). In contrast to SMT, where alignment of source and target sentence is a specific subtask, attention enables a NMT system to learn the alignments alongside with translation probabilities. The attention mechanism is explained in detail in Section 3.6.

Another contribution of Bahdanau et al. (2014) is the application of Bidirectional RNNs (BiRNNs) in the encoder. These RNNs read in the source sequence twice, once in original order and once in reversed order combining the outputs. This way, the hidden states computed at each time step comprise information about the preceding as well as about the succeeding inputs. The idea of BiRNNs is expounded in Section 3.4.3.

Table 2.1 comprises some of the results reported in the presented works showing that NMT is indeed a suitable approach to machine translation and is able to achieve acceptable results in terms of BLEU score. Especially the performance increase when applying NMT systems to the task of rescoring candidate translations produced by an SMT system is noticeable. For comparison reasons the BLEU scores of a baseline SMT system Koehn et al. (2007) is added.

Albeit the quantitative results are unfavorable for NMT in some cases where traditional SMT systems reach markedly higher scores, qualitative analyses show the capabilities of the encoder to capture the semantics of the source sentence. Figure 2.1 shows an example from Sutskever et al. (2014) where it is evident that the encodings of the source sentences are good representations of their semantics and invariant with regard to topicalization and passive versus active voice.

Additionally, the commonly used BLEU score is not an optimal measure. It does not take into account intelligibility of the translated sentences which is a noticeable strength of the presented NMT systems.

It is to note that for NMT the results achieved with systems applying RNNs surpass the

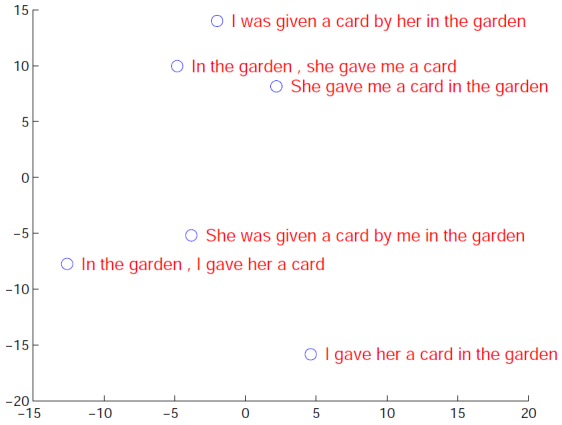


Figure 2.1: Visualization of sentences in a two-dimensional space according to their vector representation generated with the encoder of the translation system of Sutskever et al. (2014). The actual encodings of dimension 1000 are reduced in dimensionality by Principal Component Analysis (PCA).

ones obtained with convolutional approaches. This suggests that these type of NNs is inherently more suited for the task of machine translation.

2.2 Parallelization of Optimization

Apart from the amount and quality of training data, the performance of any machine learning approach, and thus NNs and NMT, is subject to the applied optimization algorithms to a great extent. The most common approaches for training NNs are gradient based methods, Stochastic Gradient Descent (SGD) and its variants being the de facto standard. A presentation of the most prominent derivatives of SGD can be found in Section 4.

As mentioned in Section 1, the training times of NNs in general and NMT systems in particular can become considerably long. This work is focused on data parallel approaches in the sense of asynchronous versions of gradient based optimization algorithms. Model parallel approaches are not considered in this work. Unfortunately, parallelization of the standard optimization algorithms is not trivial as they are inherently sequential. This section will give a short overview of works presenting asynchronous gradient based procedures. The parallel versions used in this work are then explained in more detail in section 4.

Langford et al. (2009) propose a possibility parallelizing SGD by computing gradients for different batches of training data on several processors and applying delayed updates to the parameters. These updates are then executed in order. While the parameters get written to with a small lag, processors can read parameters at any time. A proof of the functioning of the algorithm is supplied and the authors show that it is suitable for an email classification task. An important observation that was made is that less complex problems are less suited for speedups by parallelization, as the parallelization overhead surpasses the performance gains.

Taking this idea of parallelizing SGD a step further brings forth the HOGWILD! algorithm proposed by Recht et al. (2011). In HOGWILD!, the parameters are also shared among processors which calculate gradients for different batches of data. In contrast to the algorithm of Langford et al. (2009) the updates are applied to the shared parameters without any locks. Based on the fact that many machine learning tasks are applied to sparse data, the possibility of overwriting updates made by another processor is diminished to a negligible level. Thus, HOGWILD! converges with the same number of updates as SGD but due to the lock-free update policy, this convergence is sped up linearly with regard

to the number of concurrent processors. With a fairly strong sparsity assumption, Recht et al. (2011) give a convergence proof for HOGWILD!. Empirical studies show success of this approach for several problems.

The performance of SGD is subject to a sensible choice for its learning rate. The ADAGRAD algorithm (Duchi et al., 2011) proposes a revision to this fact by introducing an individual learning rate for each parameter which scale according to their associated entropy. ADAGRAD is explained in detail in Section 4.1.2. A follow-up study also assuming data sparsity presents an asynchronous version of ADAGRAD named ASYNCHADAGRAD. Introducing this algorithm yields another asynchronous optimization algorithm called called ASYNCHDA as by-product, which is based on the Dual Averaging method (Nesterov, 2009). Proofs are provided for the algorithms attaining linear speedups with respect to the number of processors. However, with the update scheme of ASYNCHADAGRAD utilizing more information at each update, the performance is further improved. The successful application of the algorithms is demonstrated in three experiments which turned out in favor of ASYNCHADAGRAD. Furthermore, effectively identical performance for ASYNCHDA and HOGWILD! is reported.

Several methods have been proposed to reduce the variance in SGD. Reddi et al. (2015) introduce a unifying framework for these methods which additionally allows them to be carried out in parallel. Among standard SGD the algorithms ingrained into this framework are Stochastic Average Gradient Descent (SAG; Schmidt et al., 2013), Stochastic Variance Reduced Gradient Descent (SVRG; Johnson and Zhang, 2013), Stochastic Average Gradient Adaption (SAGA; Defazio et al., 2014) and Semi-Stochastic Gradient Descent (S2GD; Konecnỳ and Richtárik, 2013) which have shown to outperform SGD. Another algorithm called Hybrid Stochastic Average Gradient Descent (HSAG) working in the same framework is also presented by Reddi et al. (2015). A general proof of convergence of the converted approaches is given assuming sparsity of data. In an experiment, SVRG and SGD parallelized on 10 cores are compared. Along the lines of Recht et al. (2011), a lock-free implementation of SVRG is also tested. It is shown that locking slows down training and lock-free SVRG distinctly outperforms SGD in various settings. However, the effect of parallel processing is not assessed exhaustively.

The listed methods generally make more or less strict sparsity assumptions about the processed data. Data for many NLP tasks like MT is often represented as one-hot vectors which fulfill these sparsity requirements and thus application of the presented algorithms to NMT comes naturally.

Another assumption common to the analyses of the listed works is convexity of the problems to solve. However, in real world applications, convexity is mostly not given. Empirical studies show that the algorithms nevertheless perform well and the convexity assumption is negligible.

This work focuses on the HOGWILD!, ASYNCHDA and ASYNCHADAGRAD algorithms which are explained more exactly in section 4.2. These algorithms were tested on optimizing an MLP used for a handwritten digit recognition task in a preliminary experiment (Section 5) and on training an NMT model for an English-to-French translation task (Section 6).

3 Neural Machine Translation

This section will expound the approach of NMT by gradually pointing out the used ideas of NN theory. First, the basic concepts of NNs and deep learning are revisited, giving a short explanation of feed-forward networks. The two major approaches to eliminate the limitations concerning the length of input and output sequences broached in Section 2, namely *convolution* and *recurrence*, are then explained. The focus is on RNNs since they achieved the best results for the task of translation and are applied in the experiments conducted for this work. Afterwards, the encoder-decoder architecture as proposed by Cho et al. (2014b) is depicted. This section completes with a presentation of the attention mechanism as shown in Bahdanau et al. (2014), which improved NMT models significantly, and a description of how the actual translation are sampled from the output of an encoder-decoder network.

3.1 Feed-forward Neural Networks

NNs have shown great success in different applications and deep learning is commonly used in academic research and commercial enironments. While being a relatively old invention (McCulloch and Pitts, 1943), NNs have swirled up the machine learning community only in the last decade when the amount of data and computational power constantly grew, making the resulting models more powerful (Goodfellow et al., 2016). The success of NN models is heavily based on their size and amount of data trained on while computation nowadays is still feasible. The underlying concept of NNs is inspired by the computation mechanism in the brain which coined the name Neural Networks.

In the context of NNs data is represented in the form of vectors. To represent the input words of NLP applications as vectors, the words in the sentences are transformed into so-called *one-hot vectors*. In NLP, one-hot vectors are vectors of dimensionality $|V|$ where $|V|$ is the size of the vocabulary and the values on each dimension are all 0 except for a single 1 uniquely identifying the corresponding word.

The units in a NN which are often called neurons associate weights with each dimension of these vectors. To compute the output of a neuron, an activation function is applied to the weighted sum of the inputs. Several neurons taking an input and producing a fixed number of outputs can be stacked to form a layer. By using the outputs of the units in one layer as input to units in a succeeding layer, a network with several layers can be spanned. If there is no circularity in the connections, one speaks of a *feed-forward* NN. Figure 3.1 shows an example of a modest network with input dimension 3 and output dimension 4. The shown network has two intermediate layers which are called *hidden layers*. When every neuron in one layer is connected to every neuron of the succeeding layer, they are referred to as fully-connected. The term *deep learning* stems from the fact that a network with several hidden layers is called a *deep network*.

To compute the output of a network, the input gets propagated through the different layers where the transformations explained above are performed until eventually the values on the last layer are outputted. These computations can be expressed as matrix multiplications. A network with just one output layer can thus be formulated as

$$y^{(1)} = \sigma^{(1)}(W^{(1)}x + b^{(1)}) \tag{1}$$

Here, x is the input vector, W is a $M \times N$ matrix containing the weights. Here N is the input dimension and M is the output dimension. b is an optional but often useful bias vector. The function σ resembles an *activation function* which is applied element-wise

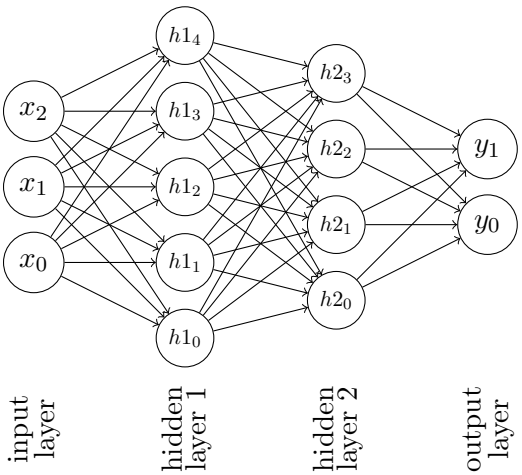


Figure 3.1: Feed-Forward Neural Network with input dimension 3, two hidden layers with 5 and 4 hidden units respectively and output dimension 2.

to its argument. Popular activation functions which also find application in many NMT system are listed in Section 3.2.

More layers can be added, wrapping the result of the previous layers in a similar function.

$$y^{(2)} = \sigma^{(2)}(W^{(2)}y^{(1)} + b^{(2)}) \tag{2}$$

The dimensions of the different layers are only restricted in that they must match their surrounding layers. The choice of activation functions is only limited by the algorithm applied for training the network. General gradient based methods as listed in Section 4 require differentiable activation functions or functions with subderivatives. The procedure of stacking layers can be continued any number of times, bringing forth arbitrarily large networks. While the number of layers influences the depth of the network, the dimension of these layers is referred to as breadth or width of a network.

While a network with a sufficiently large single hidden layer is capable of approximating any function, it may be unfeasible to train the corresponding network (Goodfellow et al., 2016). Additionally, with minor constraints for the activation functions and sufficiently many hidden units, it has been proven that feed-forward networks can approximate a very wide variety of functions (Hornik, 1991). In general, the architecture of a NN is subject to the actual learning problem and the applied learning algorithm.

Depending on the output size, NNs can be applied to different problems. A one-dimensional output can be used for regression tasks by returning the plain value or binary classification by returning the sign of the value. k-dimensional outputs can be used for multi-class classification by choosing the dimension with the highest output value. If the output values sum up to one, the output can be interpreted as probability distribution from which the outputs can be sampled.

In contrast to other machine learning algorithms, the hidden layers in NNs in a sense learn to encapsulate task specific information from the plain input values. Handcrafting features thus becomes obsolete. However, this comes at the cost of potential computational overhead because the size of the network could exceed the minimally needed size while there is no way of predetermining this.

3.2 Activation Functions

Choosing non-linear activation functions makes NNs the powerful tool they are. Without these functions, the performed transformations of the input correspond to a simple linear mapping from one dimension onto another. This mapping could also be computed in a single vector-matrix multiplication making a deep structure with more several hidden layers obsolete.

Three prominent examples of activation functions are listed below which are also relevant to the implemented system, namely the *tanh*, *sigmoid* and *softmax* functions. There is a wide range of other activation functions which are not displayed here.

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{3}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

$$softmax(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \tag{5}$$

The softmax function is somewhat special in that its output on each dimension is dependent on all dimensions of the input vector. The values of the output vector add up to 1, generating a probability distribution. Therefore, softmax units are often used on the output layer.

3.3 Convolutional Neural Networks

Feed-forward networks as depicted above are inflexible in that they require inputs with certain dimension and also produce outputs of fixed size. This is impractical for many applications with input and output of variable length. One way to overcome this shortcoming also affecting NMT is the technique of convolution. While being intrinsically suited for image processing (Xu et al., 2015), Convolutional Neural Networks (CNNs) also find application in the area of NLP. For example, Cho et al. (2014a) and Kalchbrenner and Blunsom (2013) applied convolutional algorithms to counteract the problem of variable length input and output for NMT.

The weight matrices in CNNs, often called kernels or filters, can be viewed as feature detectors which are slided over the input. At each position a convolution operation is applied which can be defined in various ways. In the case of Kalchbrenner and Blunsom (2013), the values on each dimension of the input get multiplied with a weight in the corresponding cell of the kernel. The resulting matrix then gets transformed into a vector by component-wise addition of its columns. For a kernel of width i set to 3 and an input matrix M the convolution operation is defined by

$$K^i_{:,1} \odot M_{:,a} + K^i_{:,2} \odot M_{:,a+1} + K^i_{:,3} \odot M_{:,a+2} \tag{6}$$

where a indicates the current position in the input and $\odot$ denotes component-wise multiplication. As this requires a matrix as input, Kalchbrenner and Blunsom (2013) transform the words in the input sentence into a dense vector representation. The sequence of these vectors forms a sentence matrix. The kernel is an $e \times i$ matrix where e is the size of the vector representations of the words. This convolution is then applied repeatedly until

the outputs size is reduced as desired. By applying the convolutions in this manner, a structure similar to a parse tree of the sentence is built.

In the first system, the desired vector representing the whole input sentence is obtained by convolutions with increasing values for i starting with 2 up to maximally $\sqrt{2N}$, where N is the length of the longest sentence considered, or until the size of the input is not sufficient for another convolution. In cases where more than one vector is left after these operations, an additional weight matrix of the same size as the input is applied with the same convolutional operation. The second system only applies two such convolutions with i set to 2 in the first and to 3 in the second run such that a list of vectors representing the four-grams of the input sentence is generated.

The generation of translations from these condensed representations of input sentences is discussed in Section 2.1 and is not considered here.

Another NMT system using convolution is presented in Cho et al. (2014a). A kernel of width 2 is applied to each neighboring row pairs in the sentence matrix until the input sentence is condensed into one single vector. A peculiarity is the convolutional layer introduced in this work which implements a special gating mechanism which allows information to flow through to the next level in a specific way. The information to pass on is computed as a weighted sum of each of the two vectors it is applied to and a combination of both inputs.

Other applications of CNNs often implement additional properties. For instance, it is possible to apply not only one but several kernels to the input which are trained to detect different features of the input. The number of kernels is referred to as depth of the CNN. So-called *pooling* operations are often used reducing the size of its input. Pooling works by taking a part of the input and joining the values by a certain scheme. The most prominent example is *max-pooling* which simply returns the maximum value of the part of input it is applied to. The *stride* of a convolution is the adjustable step size the kernel is moved along the input. Especially for image processing tasks, it is common to *pad* the input with zeros at its edges and such that the convolution operations can be applied prematurely. However, these features did not find their way into works using CNNs for NMT which are considered in this thesis.

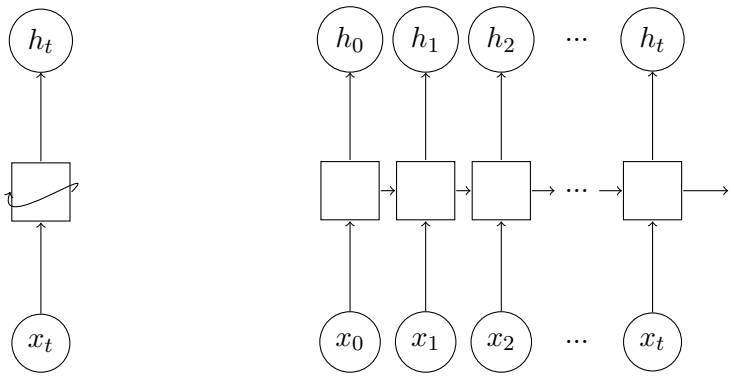
3.4 Recurrent Neural Networks

Another approach working around the requirement of fixed size input is introduced in Elman (1990). With RNNs, sequences of input can be processed by the units which not only apply weights to the current part of the input but also incorporate the hidden states generated for the previously processed input. This way, at each position, the whole history of the already processed sequence is taken into account. Hence, RNNs are very well suited for sequence processing tasks in which the order of the sequence matters. It is possible to emit a vector at each time step for *many-to-many* applications or to output only the last vector which then is a representation of the whole input sequence for *many-to-one* tasks. In reversed order, RNNs can be used to generate sequences from a single input resulting in a *one-to-many* architecture. In one-to-many applications, the outputs generated at each position are either conditioned only on the previous output, or the previous output and the initial input.

The generic recursive equation for computing the hidden states of a RNN is

$$h_t = f(x_t, h_{t-1}) \tag{7}$$

where h_t denotes the hidden state calculated by the RNN at time step t. The function



(a) Recurrent unit (b) Unrolled recurrent unit

Figure 3.2: Recurrent unit in compact and unfolded form

f is a non-linear function which can become arbitrarily complex. A simple RNN can be formulated as

$$h_t = \sigma(W_h h_{t-1} + W_x x_t) \tag{8}$$

with an activation function σ and two weight matrices W_h and W_x applied to the previous hidden state h_{t-1} and the current input x_t respectively. The values of the applied weight matrices are jointly learned when training the network. When more complex recurrent units with several weights are trained, these weights also get adjusted jointly during fitting of the model. h_{t-1} is usually initialized with a vector containing zeros in all cells for the first time step as there is no information available from any preceding inputs. Figure 3.2 shows an illustration of the standard RNN architecture. Analogous to feed-forward nets, several recurrent layers can be stacked to obtain a deep RNN.

Despite the capability of transmitting information throughout the whole network, in practice, basic instances of RNNs fail to convey information encountered far off in the past of the input sequence. To solve this issue, units implementing gating mechanisms which allow passing information in more sophisticated ways have been proposed. The most famous version of the recurrent unit are the LSTM units explained in the next section. Following that, a description of another gated recurrent unit simply called GRU introduced in Cho et al. (2014b), which also tackles this problem, is given. The later unit is used in the system implemented for this work.

3.4.1 Long Short-Term Memory Units

The LSTMs presented in Hochreiter and Schmidhuber (1997) have achieved excellent results in several fields. These units are based on the idea of a persistent memory flowing and being updated throughout processing a sequence. This memory is managed in the *cell state* C_t and is altered by different gates. The first gate, the *forget gate*, calculates the amount of existing memory to maintain or drop according to the previous hidden state and the current input. It is calculated by

$$g_{forget} = \sigma(W_{forget}[h_{t-1}, x_t] + b_{forget}) \tag{9}$$

where $[\cdot, \cdot]$ stands for the concatenation of two vectors. As the activation function σ yields values between 0 and 1, element-wise multiplication with the cell state causes irrelevant information to be dropped and important information to be retained.Values computed

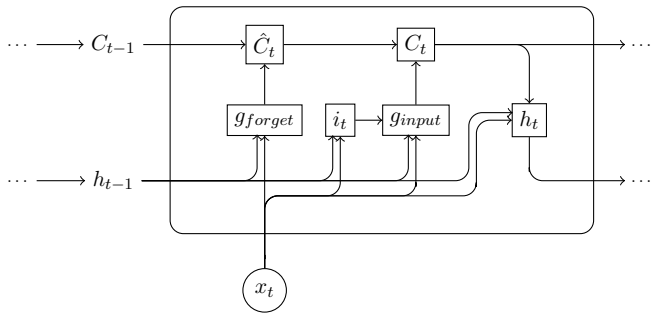


Figure 3.3: Schematic illustration of a Long Short-Term Memory unit according to Hochreiter and Schmidhuber (1997). Inspired by Olah (2015).

in the forget gate near 0 indicate that an information is rather irrelevant in a particular setting and values near 1 respectively indicate important information to keep.

$$\hat{C}_t = C_t * g_{forget} \tag{10}$$

The *input gate*, calculated as follows, then first determines how much information to add

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{11}$$

and which information exactly to add to the cell state

$$g_{input} = i_t * (tanh(W_{input}[h_{t-1}, x_t] + b_{input}) \tag{12}$$

The cell state then is updated with the new information

$$C_t = \hat{C}_t + g_{input} \tag{13}$$

An additional *tanh* layer is applied to the concatenation of the previous hidden state and the current input. Element-wise multiplication with the cell state after this state is run through a *tanh* function then yields the current hidden state h_t.

$$h_t = (\sigma(W_o[h_{t-1}, x_t] + b_o)) * tanh(C_t) \tag{14}$$

The activation functions σ can be chosen differently. The weight matrices W_{forget}, W_i, W_{input} and W_o as well as the biases b_{forget}, b_i, b_{input} and b_o are jointly adjusted throughout the training of the network. Figure 3.3 shows a schematic illustration of an LSTM unit.

Two succeeding deep LSTM networks with 4 layers each are used for NMT in Sutskever et al. (2014) with great success.

3.4.2 Gated Recurrent Units

Another recurrent unit with gating mechanisms applied successfully to NMT tasks is introduced in Cho et al. (2014b). The proposed GRUs, are similar yet more simple than the LSTM units presented in the previous section. There is no cell state, rather two gates manage which and how much memory flows through from the last time step solely based on the previous hidden state h_{t-1} and the current input x_t. In a reset gate which corresponds to the forget gate of LSTM units, the previous hidden state and the current

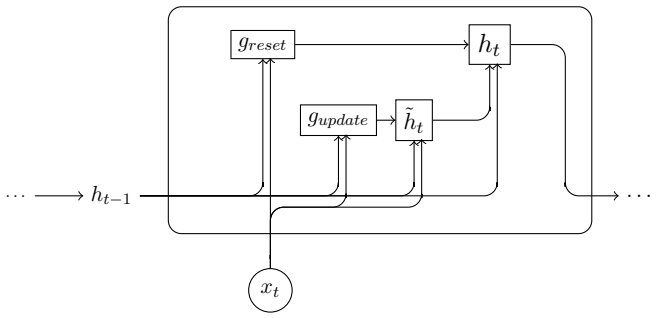


Figure 3.4: Schematic illustration of a Gated Recurrent unit according to Cho et al. (2014b)

input get weighted according to two weight matrices W_{update} and U_{update}. The outcomes are summed element-wise and an *sigmoid* is applied which results in a vector indicating the amount of memory to drop. This gating mechanism is different from the one in LSTMs where the inputs are concatenated and are weighted by one single matrix.

$$g_{reset} = sigmoid(W_{reset}x_t + U_{reset}h_{t-1}) \tag{15}$$

Similarly, the update gate calculates the amount of new information to add

$$g_{update} = sigmoid(W_{update}x_t + U_{update}h_{t-1}) \tag{16}$$

and the actual new information to add is computed by

$$\tilde{h}_t = tanh(Wx_t + [U(g_{update} \odot h_{t-1})]) \tag{17}$$

where $\odot$ denominates element-wise multiplication, and addition is also applied element-wise. The hidden state at time step t is then the sum of the previous hidden state and the proposed hidden state $\tilde{h}_t$ weighted by g_{reset}.

$$h_t = g_{reset}h_{t-1} + (1 - g_{reset})\tilde{h}_t \tag{18}$$

This way, the unit drops irrelevant information and only stores new information considered relevant allowing a compact representation of the input sequence. Figure 3.4 shows a schematic representation of the unit.

Cho et al. (2014b) state that this kind of recurrent units performs considerably better for NMT than a standard recurrent unit with a tanh activation function.

3.4.3 Bidirectional Recurrent Networks

Standard RNNs read in a sequence of inputs one by one retaining information about previously read inputs and thus also their order. In some cases, it may be desirable to not only have information about the previous time steps but rather additional information about succeeding inputs. This can be achieved by also running the same sequence through an RNN in reversed order and combining the outputs of both RNNs at each time step. This forms a structure called Bidirectional Recurrent Neural Network (BiRNN; Schuster and Paliwal, 1997). There are various ways to combine the resulting sequence of outputs in many-to-many tasks or the two single outputs in a many-to-one application. BiRNNs are not suitable for one-to-many applications as the sequence only is generated. The canonical

way to combine the outputs is concatenation, but there are various possibilities. In the system implemented for this work, the outputs are concatenated and the mean of outputs over the whole sequence is used as representation thereof.

As mentioned in Section 2.1, Sutskever et al. (2014) noticed a performance increase of their NMT system when reversing the input sentences. This finding reinforces the rationale of a backward pass over the input sequence having a positive effect on the outputs quality. Wile their findings hold for the language pairs tested on, a combination of forward and backward passes can further increase quality. Generally, in language, the meaning of a specific word can be inferred better from the whole context, i.e. preceding and succeeding words. An NMT system using a BiRNN for encoding the input sentence is used in Bahdanau et al. (2014) and also in the work in hand.

3.5 The Encoder-Decoder Architecture

The previous section made it obvious that RNNs are suitable to override the requirement of NNs of fixed-size inputs. However, when applied to a sequence, RNNs are only capable of generating sequences of the same length in a many-to-many application or a single output in a many-to-one setting. In MT and various other sequence-to-sequence tasks, the input and output sequences differ in length. The encoder-decoder architecture as proposed in Cho et al. (2014b) combines two RNNs to overcome this limitation.

A first RNN, the encoder, condenses a sentence of variable length into a vector representation c of fixed length often called the *context vector*. This vector is a semantically meaningful representation of the input sentence (see Figure 2.1 for an example). Starting from this representation, a second RNN, the decoder, generates the target sentence word by word until a special end-of-sentence token is generated or a specified maximum length is reached. The generation of each word is conditioned on the context vector, the previously generated word and the previous hidden state, similar to the recurrent language models described in Mikolov et al. (2010). While the lengths of the sequences certainly correlate, this way, the length of the output is not dependent on the length of the input. Figure 3.5 shows a schematic illustration of the architecture. It is possible for the encoder and the decoder to consist of deep RNNs with several layers.

Adapting this general framework, it is also possible to apply a CNN as encoder as in Cho et al. (2014a) or the first system in Kalchbrenner and Blunsom (2013). Using a CNN as decoder requires evaluation of the output sentence length beforehand as the generation of a special token is not easily feasible. A suitable CNN expanding the encoding does not sequentially generate words but rather generates the outputs simultaneously. Therefore, the token indicating termination could be encountered anywhere in the sequence and a way to handle this is required. Also, the results of previous work discourage the usage of CNNs for NMT.

For training an encoder-decoder network, both the source and the target sentences are known and inputted as sentence pairs. The goal of training then is to maximize the conditional probability of the target sentences given the corresponding source sentences. As the equation form of the encoder-decoder is end-to-end differentiable, training can also be performed with the gradient based methods described in Section 4.

The encoder-decoder architecture allows preceding and succeeding layers. An often used embedding layer transforms the words represented as one-hot vectors into more dense continuous representation. The system implemented for this work includes an initial and a final embedding layer. It is

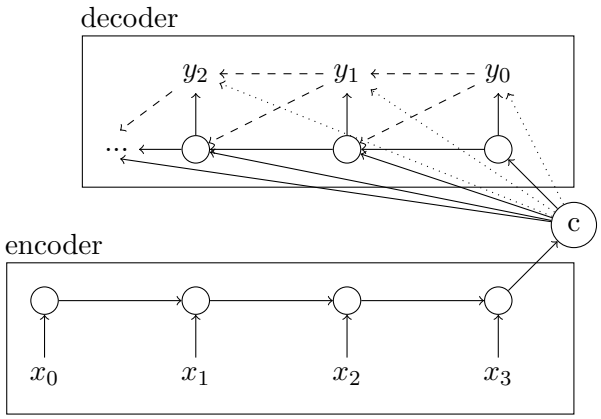


Figure 3.5: Schematic depiction of the encoder-decoder architecture

3.6 The Attention Mechanism

Despite RNNs being capable to convey information seen many time steps before, in practice they often fail to do so. Even LSTMs and GRUs which are designed to counteract this exact shortcoming show vulnerability when applied to long sequences. This is due to the size of the context vector not sufficing to capture all information present (Bahdanau et al., 2014). The so-called *attention mechanism* (Graves, 2013) poses a possibility to work around this problem. This method is inspired by the visual attention of humans as well as animals and is visualized accessibly in Xu et al. (2015). The attention mechanism is implemented in the system of Bahdanau et al. (2014) and improved their NMT system considerably. It is also implemented in the system used for this work.

In contrast to the encoder-decoder without attention mechanism, the generation of each word in the decoder is not conditioned on a single context vector c but rather on several vectors c_i for each part of the input of length T. These vectors are computed at each time step i of the generated sequence as a weighted sum of all hidden states h_j produced by the encoder and sum up to one over all time steps j of the input sequence.

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \tag{19}$$

The attention weights α indicate proportional importance of the jth part of the input for the part of the output sequence to generate next. ß$alpha_{ij}$ is defined by

$$\alpha_{ij} = \frac{exp(e_{ij}}{\sum_{k=1}^{T} exp(e_{ik})}) \tag{20}$$

where e_{ij} is computed as

$$e_{ij} = a(s_{i-1}, h_j) \tag{21}$$

The function a is then incorporated into the training process in the form of a feed-forward network which is jointly trained with the rest of the network (Bahdanau et al., 2014). s_{i-1} represents the previous hidden state produced by the decoder RNN. These weights emphasize the points in the input containing the most important information for the next output to generate. This application of attention is called *soft attention*. With *hard attention* only one of the encoder's hidden states is regarded, namely the one with the highest attention weight. That is, the generation of the output sequence at each time step

is conditioned only on one part of the input. In a way, the attention mechanism corresponds to the component of SMT systems aligning the words in the source and target sentences.

In contrast to the intuitive interpretation of attention, which means blanking out irrelevant information because one part of the input attracts all attention, the attention mechanism assigns importance to all parts of the input for each generated output. In accordance to the length of input and output, this becomes computationally expensive.

3.7 Sampling Translations

After the input is passed through the encoder, a representation of the input in form of a single context vector or several context vectors when applying the attention mechanism is obtained. From this representation the decoder generates the target sequence one by one until a special token indicating termination is generated or a certain maximum length is reached. In NMT, the token determining termination is the end-of-sentence token and the outputs are probability distributions over the target vocabulary generated by a final transformation with a softmax activation function. To render the most suitable translation from this sequence of distributions, most implementations of the encoder-decoder use beam search (see for example Cho et al., 2014a).

Beam search generates hypotheses by calculating the probabilities of each word in the target vocabulary and only keeping the n most probable ones at each time step. Since the generated words are conditioned on the previously generated ones, several possible paths are simulated. Whenever the end-of-sentence token is generated, the corresponding path is removed from the hypotheses and added to a set of possible translations with its score (log probability). This procedure is run until all hypotheses ended in the end-of-sentence token or the maximum length is reached. In case of translation, the translation with the highest score is returned. Sutskever et al. (2014) report of a beam size of 1 already performing well and a beam size of 2 exhaustively exploiting the advantages of beam search.

4 Optimization of Neural Networks

Like other machine learning approaches, NNs rely on finding the parameters that make the model suit the underlying data well. This is achieved by optimization, which is often referred to as learning, training or fitting a model. In contrast to pure optimization, for NN training a perfect solution is not desirable. A model adapting to the training data perfectly and thus also possible noise and variance within this data may fail to generalize on unseen data. This phenomenon is referred to as *overfitting*. Apart from the quality and amount of data used for training, a considerable amount of success is based on the applied optimization algorithm. Owing to the large number of parameters and the non-linear nature of NNs, analytical solutions are unfeasible. Hence, the prevalent approach to train NNs are iterative gradient based methods.

For training NNs with gradient based methods, the output of the network is computed for a set of inputs where the true outcome is known. Based on the difference between the computed and desired output values, an error can be estimated. By propagating this error back through the network, the gradients for all parameters can be computed efficiently using the *backpropagation* algorithm (Rumelhart et al., 1988). The weights are then updated according to the computed gradients and the applied optimization algorithm for several iterations. This procedure is repeated until eventually a stopping criterion is met.

This section lists the most commonly used gradient based algorithms for training NNs. Starting with SGD, this section will move to more advanced adaptations thereof. Explanations of the often used *momentum method* (Qian, 1999) and Nesterov Accelerated Momentum (NAG; Nesterov, 1983) are not given as these algorithms were not tested in the experiment conducted for this work.

Afterwards, asynchronous optimization schemes are presented with a focus on the ones implemented for this work. This section will conclude with a comprehensive list of different strategies to improve training, that are applicable to all optimization algorithms mentioned in this section.

4.1 Gradient Based Optimization Algorithms

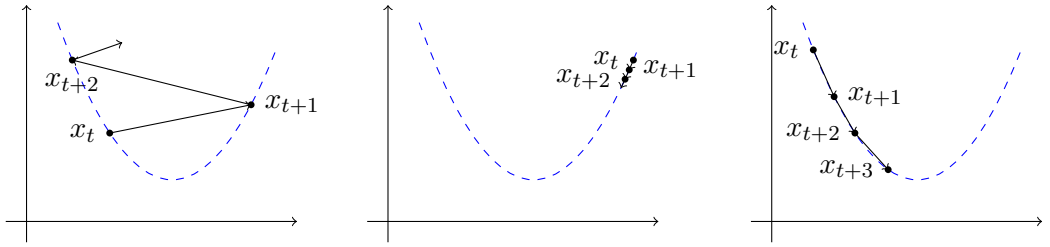
Training an NN is a minimization problem. A cost or loss function l which is dependent on the parameters of the model θ is defined over the training set.

$$\min_{\theta} l(f(x, \theta), y) = \min_{\theta} l(\hat{y}, y) \tag{22}$$

where f stands for the transformation of data from input x to predicted output $\hat{y}$ by the NN and y is the known true output value. The goal of the training is to set θ such that l yields minimal cost for a set of training samples (x, y). There are different ways to define the cost function. Based on the problem, some are more suited than others. Among many others, prominent examples are the *mean squared error* (MSE) and *negative log likelihood* (NLL) functions.

$$MSE = \frac{1}{N} \sum_{i} (\hat{y}_i - y_i)^2 \tag{23}$$

$$NLL = -\sum_{i} log(p(y_i|x_i)) \tag{24}$$



(a) Too high learning rate (b) Too small learning rate (c) Suitable learning rate

Figure 4.1: Visualization of optimization steps finding the minimum value in a two-dimensional space with different learning rates

NLL does not depend on the predicted class itself but rather its probability.

The most common approach optimizing the parameters of NNs are gradient based methods. This kind of algorithm makes use of the fact that the gradient of a function yields towards points where its function value increases. In the case of cost functions, points of higher function values are identical to points of higher cost. Changing the parameters in the negative direction of the gradient is thus tantamount to obtain lower cost afterwards. Thus, when using such algorithms, it is necessary that the cost function is derivable. Relying on this same idea there are several gradient based methods which update the parameters differently.

The update scheme of the standard gradient descent algorithm is defined by

$$\theta_{t+1} = \theta_t - \epsilon g \tag{25}$$

where ϵ is a step size often called the learning rate and g is the gradient computed for l with the current parameters

$$g = \nabla_\theta l(f(x, \theta), y) \tag{26}$$

In case of the gradient being computed for more than one sample, it is averaged over all these n samples

$$g = \frac{1}{n} \nabla_\theta \sum_i l(f(x_i, \theta), y_i) \tag{27}$$

Figure 4.1 is a coarse illustration of updates performed by a gradient based method aiming to find the minimum of a function defined in two-dimensional space with different learning rates. On the one hand, if the learning rate is set too high, the parameters may get updated such that the applied updates result in higher cost (Figure 4.1a). On the other hand, too small learning rates may result in a unnecessarily long time needed for the algorithm to converge (Figure 4.1b). Thus, a sensible choice for the learning rate is crucial to the performance of gradient based approaches. Based on the assumption that the steps should be smaller once a minimum is approached, to find the best value, a common strategy is decreasing the learning rate as training advances.

The major difference between the algorithms presented hereafter lies within computation of the gradients and how the parameter updates are performed. Some of the presented algorithms diminish or completely remove the effect of poorly chosen learning rates.

Algorithm I SGD

Require: initial parameters θ_0
Require: learning rate ϵ
1: **for** t **do** in 1,2,3,...,T
2: sample x from training data ▷ x is either a single sample or a batch of samples
3: compute gradient g with that sample
4: apply update $\theta_t = \theta_{t-1} - \epsilon g$
5: apply learning rate decay ▷ optional step performed according to a predefined schedule
6: **end for**

4.1.1 Stochastic Gradient Descent

Since computing the exact gradient over the whole data set is inefficient, SGD (Algorithm I) picks a sample from the training set and calculates the gradient based on this sample. The update rule for this algorithm is the same as for the original gradient descent algorithm (Equation 25).

In an adapted variant of SGD, instead of one sample, batches of certain size are sampled to get a more precise estimate of the gradient (Equation 27). This method originally is called mini-batch gradient descent but the conceptual distinctions are blurred. It is to note that processing batches does not slow down training a lot as the matrix multiplications can be combined. This effect can be observed mainly for computations on GPU. This way, the batch size does not only reduce variance in gradient estimates but a properly set batch size makes advantage of the computational resources optimally (Goodfellow et al., 2016).

In practice, stochastic and mini-batch methods are robust and yield decent gradient estimates and thus good parameter updates while saving a considerable amount of computations compared to the computation of gradients over the whole data set (Goodfellow et al., 2016).

4.1.2 AdaGrad

Standard SGD is prone to poorly chosen learning rates. Aside from that, in standard gradient descent, all parameters are updated according to the same learning rate. An approach to reduce the impact of the initial learning rate is proposed by Duchi et al. (2011). The rationale of ADAGRAD (Algorithm II) is to scale parameters according to their importance by introducing individual adaptive learning rates for each parameter. Rarely updated parameters are altered at a higher rate assuming they are more predictive. Following the same idea, frequently updated parameters are not changed that much. This way, the updates are still dependent on an initial learning rate but it is not necessary to adapt it over time as it is with SGD. The gradient g is calculated the same way as in SGD (Equations 26 and 27) while the update rule in ADAGRAD changes to

$$\theta_t = \theta_{t-1} - \frac{\epsilon}{\sqrt{G_t + \eta}} \odot g_t \tag{28}$$

$\odot$ means element-wise multiplication. ϵ here is a vector of the same dimension as the number of gradients in g. Thus, the division which is also applied element-wise generates the appropriately weighted learning rate for each gradient. η is a small smoothing term establishing numerical stability avoiding divisions by zero. While the update still is dependent on an initial learning rate for ϵ, the effect of poorly chosen values is reduced and it is not necessary to adapt ϵ over time. To keep track of the updates a parameter is

Algorithm II AdaGrad

Require: initial parameters θ_0
Require: initial learning rate ϵ, smoothing term η
1: $G_0 = 0$
2: **for** t **do** in 1,2,3,...,T
3: sample x from training data ▷ x is either a single sample or a batch of samples
4: compute gradient g with that sample
5: accumulate gradient history $G_t = G_{t-1} + g^2$
6: apply update $\theta_t = \theta_{t-1} - \frac{\epsilon}{\sqrt{G_t + \eta}} \odot g$
7: **end for**

Algorithm III RMSprop

Require: initial parameters θ_0
Require: initial learning rate ϵ, decay rate ρ, smoothing term η
1: $G_0 = 0$
2: **for** t **do** in 1,2,3,...,T
3: sample x from training data ▷ x is either a single sample or a batch of samples
4: compute gradient g with that sample
5: accumulate gradient history $G_t = \rho G_{t-1} + (1 - \rho) g^2$
6: apply update $\theta_t = \theta_{t-1} - \frac{\epsilon}{\sqrt{G_t + \eta}} \odot g$
7: **end for**

subject to, the sum of squares of gradients G_t contains the sum of the squared previous gradients

$$G_t = \sum_t g_t^2 \tag{29}$$

To efficiently compute G_t, it can also be updated in each iteration.

$$G_{t+1} = G_t + g^2 \tag{30}$$

where g is the gradient computed in the t-th iteration.

4.1.3 RMSprop

AdaGrad monotonously shrinks the learning rate eventually to virtually zero and training may come to a standstill. RMSprop (Algorithm III; Tieleman and Hinton, 2012) is less conservative in this respect.

In contrast to AdaGrad, the sum of squared gradients is weighted by a decay rate ρ and hereby a weighted running average is computed.

$$G_t = \rho G_{t-1} + (1 - \rho) g^2 \tag{31}$$

The algorithm thus adapts to specifics in gradients computed with the current parameters better. While G is calculated differently, the update rule in RMSprop stays the same as in AdaGrad (see Equation 28).

Algorithm IV ADADELTA

Require: initial parameters θ_0
Require: decay rate ρ, smoothing term η ▷ ρ is usually set to values ~ 0.9
$G_0 = 0$
$U_0 = 0$
for t **do** in 1,2,3,...,T
  sample x from training data ▷ x is either a single sample or a batch of samples
  compute gradient g with that sample
  accumulate gradient history $G_t = \rho G_{t-1} + (1-\rho)g^2$
  compute update $\Delta\theta_t = -\frac{\sqrt{U_{t-1}+\eta}}{\sqrt{G_t+\eta}}g$
  apply update $\theta_t = \theta_{t-1} + \Delta\theta_t$
  accumulate updates $U_t = \rho U_{t-1} + (1-\rho)\Delta\theta_t^2$
end for

4.1.4 Adadelta

Analogous to RMSPROP, ADADELTA (Algorithm IV; Zeiler, 2012) uses a decaying average of squared gradients to reduce the impact of gradients encountered many updates before (see Equation 31). By also taking into account a similar weighted average of previously applied updates, ADADELTA obviates the need for a manually chosen learning rate.

The squares of previously applied parameter updates $\Delta\theta_t$ are stored in the weighted average U.

$$U_t = \rho U_{t-1} + (1-\rho)\Delta\theta_t^2 \tag{32}$$

where $\Delta\theta_t$ is calculated by the following formula in which the square roots of running averages often are referred to as Root Mean Squares (RMS).

$$\Delta\theta_t = \frac{\sqrt{U_{t-1}+\eta}}{\sqrt{G_t+\eta}}g = \frac{RMS(U_{t-1})}{RMS(G_t)}g \tag{33}$$

The different subscripts of U and G make it clear that the sum of updates is only accumulated after being applied. The update rule of ADADELTA is formulated as

$$\theta_t = \theta_{t-1} - \Delta\theta_t^2 \tag{34}$$

4.1.5 Adam

The Adaptive Moment Estimation algorithm (ADAM, Algorithm V) presented by Kingma and Ba (2014) keeps track of the gradients as well as the squared gradients as weighted sums according to two different decay rates ρ_1 and ρ_2.

$$G_t = \rho_1 G_{t-1} + (1-\rho_1)g \tag{35}$$

$$S_t = \rho_2 S_{t-1} + (1-\rho_2)g^2 \tag{36}$$

These two moment estimates are then adjusted, growing in importance as the iteration number t increases.

Algorithm V ADAM

Require: initial parameters θ_0
Require: initial learning rate ϵ, decay rates ρ_1 and ρ_2, smoothing term η
$G_0 = 0$
$S_0 = 0$
for t **do** in 1,2,3,...,T
  sample x from training data ▷ x is either a single sample or a batch of samples
  compute gradient g with that sample
  accumulate gradient history G: $G_t = \rho_1 G_{t-1} + (1 - \rho_1)g$
  accumulate gradient history S: $S_t = \rho_2 S_{t-1} + (1 - \rho_2)g^2$
  compute $\hat{G}_t = \frac{G_t}{1-\rho_1^t}$ and $\hat{S}_t = \frac{S_t}{1-\rho_2^t}$
  apply update $\theta_t = \theta_{t-1} - \epsilon \frac{\hat{S}_t}{\sqrt{\hat{G}_t + \eta}}$
end for

$$\hat{G}_t = \frac{G_t}{1 - \rho_1^t} \tag{37}$$

$$\hat{S}_t = \frac{S_t}{1 - \rho_2^t} \tag{38}$$

The sum of previous gradients G can be seen as the momentum of the updates. While SGD only considers the current gradients, ADAM has a notion of a general direction of updates and thus is more resilient to divergent gradient estimates encountered for outliers within the samples. ADAM reportedly is robust to the settings and one of the most often applied optimization algorithms for NN training (Goodfellow et al., 2016).

4.2 Asynchronous Optimization Algorithms

In the algorithms presented above the newly computed gradients are dependent on parameter updates in previous iterations. Therefore, these algorithms are inherently sequential procedures. While the inner calculations in their application to NN optimization can be parallelized very well on GPUs, there exist algorithms that allow asynchronous processing of more than one sample or a batch of samples at a time. These algorithms allow parallelizing optimization on a higher level, that is, on multi-core processors or GPU clusters. This can save tremendous amounts of time for training NN models, which is beneficial regarding the massive quantity of data they are applied to.

Section 2.2 listed a few works concerned with asynchronous optimization algorithms. In this section, the three algorithms used for the experiment conducted for this work, namely HOGWILD!, ASYNCHDA and ASYNCHADAGRAD, are revisited and examined further.

4.2.1 Hogwild!

Recht et al. (2011) introduce an algorithm called HOGWILD! that carries out updates in the fashion of SGD (Algorithm I) on a number of concurrent processors. The parameters θ are shared among these processors which calculate gradients of different training samples in parallel. Each processor then applies updates to the shared parameters without any locks. Due to this lock-free approach, generally race conditions may occur and thus updates performed by other processors may get overridden. However, by assuming sparsity of data, Recht et al. (2011) give a mathematical proof of convergence as well as empirical success of the algorithm on several tasks is shown. Sparse data in this context means

that the vectors representing the data are zero-valued on the majority of dimensions. The success of the algorithm can thus be attributed to the fact that in such sparse settings, the updates only affect a small subset of the parameters and the processes access mostly different values when updating. It is important to note that in HOGWILD! individual scalar values of the multidimensional parameters θ are altered separately. Neglecting the occurrence of possible overwriting and the communication overhead between processes, HOGWILD! needs the same number of updates to converge as SGD. As it is run on several processors, theoretically, a linear speedup is achieved corresponding to the number of concurrent processors.

In NLP, data is often sparse as words are represented in one-hot vectors. The results in Section 6.5 show that HOGWILD! indeed can be applied to NMT to speed up training. It also is suitable for training less complex NNs as the results of a preliminary experiment show in Section 5.3. However, in this somewhat smaller setting, the parallelization overhead predominates the improvement of parallelization markedly.

4.2.2 AsynchDA

While developing an asynchronous version of ADAGRAD, which is described in the following section, Duchi et al. (2013) propose the ASYNCHDA algorithm. This asynchronous optimization algorithm is based on the dual averaging algorithm attributed to Nesterov (2009).

The proposed asynchronous optimization algorithms by Duchi et al. (2013) are tailored for convex problems and, similar to Recht et al. (2011), assume data sparsity. The sparsity assumption in this context can be characterized as the gradients of the loss function having less zero values than the input samples. That is, the loss function is independent of the zero values of the samples.

By transferring the optimization problem into its dual form, the parameters are updated in each iteration t by solving the inner optimization problem

$$\theta_t = \arg\max_{\theta_t \in \chi} \{\langle G_{t-1}, \theta_{t-1} \rangle + \frac{1}{\epsilon} \psi(\theta_{t-1})\} \tag{39}$$

where χ is a convex set and ψ is a regularization term. G_t here is a (dual) vector containing the sum of previously computed gradients on all processors. This step is carried out by each processor individually and only the shared sum of gradients G is read from the main process.

By only depending on the running sum of gradients, this procedure can easily be parallelized by running Algorithm VI on several concurrent processes. The only communication point between these processes resides in the last step of accumulating G (line 6 in the pseudo code). Due to associativity and commutativity of addition, the order of these operations is irrelevant. Thus, no synchronization problems are encountered.

Using the suggestion of Duchi et al. (2013), setting ψ to $\frac{1}{2}||\theta||_2^2$ and additionally neglecting the convexity assumption, Equation 39 can be solved analytically by setting its derivative to zero. The parameters θ are updated to

$$\theta_t = -\epsilon G_{t-1} \tag{40}$$

in each iteration.

However, the negligence of convexity is not expedient. The experiments carried out for this work show that the algorithm is not suitable for the problems tested on. While working

Algorithm VI AsynchDA

Require: initial parameters θ_0
Require: learning rate ϵ
Require: shared object G
1: $G_0 = 0$
2: **for** t **do** in 1,2,3,...,T
3: Read G_{t-1} and compute $\theta_t = \arg\max_{\theta_t \in \chi}\{\langle G_{t-1}, \theta_{t-1}\rangle + \frac{1}{\epsilon}\psi(\theta_{t-1})\}$ ▷ reading from shared memory
4: sample x from training data ▷ x is either a single sample or a batch of samples
5: compute gradient g with that sample
6: accumulate shared gradient history G: $G_t = G_{t-1} + g$ ▷ writing to shared memory
7: **end for**

for optimization of an MLP under certain circumstances, AsynchDA is not suitable for NMT optimization (see Sections 5.3 and 6.5).

4.2.3 AsynchAdaGrad

While Duchi et al. (2013) report near similar performance of Hogwild! and AsynchDA in empirical studies, another algorithm is introduced which yields a slightly increased speedup and improved model quality. The AsynchAdaGrad algorithm extends AsynchDA by taking into account the sum of squares of the previously encountered gradients. The updates of the parameters performed in each iteration of the algorithm are then adjusted in the fashion of AdaGrad.

The inner optimization problem (Equation 39) is reformulated as

$$\theta_t = \arg\max_{\theta_t \in \chi}\{\langle G_{t-1}, \theta_{t-1}\rangle + \frac{1}{2\epsilon}\langle x, \sqrt{S_{t-1}}\theta_{t-1}\rangle\} \tag{41}$$

By disregarding the projection onto a convex set of parameters, this can also be solved analytically. The parameters are updated by the following rule

$$\theta_t = -\frac{\epsilon G_{t-1}}{S_{t-1}} \tag{42}$$

where S is the sum of squares of previous gradients computed on all processors.

The usage of AsynchAdaGrad exhibits very fast convergence when used for optimization of an MLP in the preliminary experiment (Section 5). Since it is parallelizable like AsynchDA, this property promises considerable speedups for training different kinds of NNs. However, for optimizing an NMT system, the algorithm does not yield the anticipated success (see Section 6.5).

4.3 Strategies Improving Optimization

Apart from an annealing schedule for the learning rate as mentioned before, there are several methods enhancing the performance of the algorithms presented in this section. The strategies listed in this section are approaches to enhance the training of NNs and thus improve the quality of the resulting models. The following list is not exhaustive and represents only a few techniques commonly used to improve training of NNs. For a

Algorithm VII AsynchAdaGrad

Require: initial parameters θ_0
Require: learning rate ϵ
Require: Shared objects G and S
$G_0 = 0$, $S_0 = 0$
for t **do** in 1,2,3,...,T
  Read G_{t-1}, S_{t-1} and compute $\theta_t = \arg\max_{\theta_t \in \chi}\{\langle G_{t-1}, \theta_{t-1}\rangle + \frac{1}{2\epsilon}\langle x, \sqrt{S_{t-1}}\theta_{t-1}\rangle\}$ ▷ reading from shared memory
  sample x from training data ▷ x is either a single sample or a batch of samples
  compute gradient g with that sample
  accumulate shared gradient histories G and S: $G_t = G_{t-1} + g$ $S_t = S_{t-1} + g^2$ ▷ writing to shared memory

end for

more detailed description of the methods and further methods improving optimization see Goodfellow et al. (2016).

It is possible for the computed gradients to become quite large. To avoid updating parameters too severely, the technique of *gradient clipping* can be applied. If a gradient exceeds a certain threshold, it is decreased to a lower value and thus has not such a big impact.

The parameters of NNs are initialized randomly. While the repercussions may vanish relatively fast after a few iterations, heuristic parameter initialization schemes have shown success. *Glorot* or *Xavier initialization* (Glorot and Bengio, 2010) samples the initial weights of a neuron from an uniform distribution.

$$W_{i,j} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}) \tag{43}$$

where m is the number of inputs and n the number of outputs of the unit. While the weights are initialized randomly, Goodfellow et al. (2016) state that biases are best set to constant values. Apart from the initialization strategy used, it is important to note that units initialized with the same values yield the exact same outcome after being altered the same way throughout training and thus are redundant. Despite the dropout procedure explained later overrides the fact that the units are subject to the same changes, this effect is very small and identical initialization is to be avoided.

Training several models as in the *bagging* method can improve the outcomes by applying all models to a problem. The output then is generated by computing the predictions of all models for the input and voting on the final result.

However, since the training of several independent models in bagging is computationally expensive, the already mentioned *dropout* is to prefer. When using dropout, some of the units are randomly deactivated in each iteration by multiplying their outputs by zero. This way, the model is more independent from the outputs of individual units and thus becomes more robust. Goodfellow et al. (2016) show that bagging is almost identical to applying dropout on all layers.

For training NMT systems or other learning tasks concerned with variable length input, biasing the training batches can be beneficial. By generating samples with sentences of similar length, the variance in gradient estimates can be reduced.

As stated in the beginning of this section, optimization in machine learning differs from standard optimization as it is not considered with finding the perfect solution for a given

training data set. Rather, a good enough solution which generalizes well to unseen data is desired, that is, a solution that has low generalization error. If a model fits the training data very well but performs badly on unseen data, the model is said to be overfitted.

There are several methods to counteract overfitting, one of which is *early stopping*. The training data is partitioned into a training data set and a validation data set. A typical fragmentation is using 20 % of the training data for validation (Goodfellow et al., 2016). The validation data is held out from training and the model is tested on this data in specified intervals. Once the performance of the model calculated on the validation data set does not increase anymore, training is halted as the model begins to fit the training data too well. A common approach is to not immediately stop training but to implement a *patience*. Training is continued until the model did not improve for several validation runs. This way, destructive impact of outlying validation values is evaded.

All methods listed here regularize the training in a way. There is also another option to prevent overfitting actually called *regularization*. A regularizing term, often called penalty, is added to the cost function restricting the influence of the actual cost. Usual regularization terms are weight norms like $l1$ or $l2$ norm avoiding the parameters to become very big and influence outcomes too heavily.

5 Preliminary Experiment

In order to get a better picture of parallel computing in the context of NNs, a preliminary experiment on a more plain problem is conducted. The task of handwritten digit recognition with a simple feed-forward NN is chosen as it poses a meaningful problem with real world application scenarios while still not being too complex. The chosen task also is a canonical example for classification often used for benchmarking NN systems (see for example Ciresan et al., 2011). This experiment shows the different use cases of this work and helps to better set the later findings in context while also facilitating comparison of this work to a wide range of other works concerned with the same object recognition problem. Development benefits from this preliminary work as it allows for quicker testing and easy porting of the codes. Therefore the implementation for this experiment uses the Python library `theano`. Details of the implementation are described later in Section 6.1. Some peculiarities affecting the implementations of the asynchronous optimization algorithms are also pointed out in this subsequent section.

5.1 Data

The MNIST database[3] originally comprises 70.000 images of handwritten digits from 0 to 9 (LeCun et al., 1998). These images are represented as 28×28 matrices with real valued entries from 0 to 255 corresponding to shades of gray from white (0) to black (255). The correct labels for each of the images is the number of the digit shown in the image. The data usually is split into 60.000 samples for training and 10.000 samples for testing.

In this work, a recompiled version of the original data[4] is used. This data represents the original matrices in flattened form as $\mathbb{R}^{784}$ vectors stored as `numpy` arrays. The packaging of the data as python `pickle` file is a further convenience for this work. The division of the data differs from the original splits as 10.000 of the training samples are additionally separated into a validation data set. The results on this data are not reported here but can be found on the CD attached to the printed version of this thesis.

5.2 Experiment Setting

For this experiment, a fully-connected feed-forward MLP as described in Section 3.1 with one hidden layer comprising 1.024 units is used. The weights and biases are initialized with random values sampled from a continuous random distribution of values between 0 and 0.01. To obtain the values in the hidden layer, a *tanh* activation function is applied while a $softmax$ function computes a probability distribution over the 10 possible labels on the output layer.

The training data is shuffled once in the beginning and the order then is maintained throughout the whole training process. In each iteration, batches of size 32 are sampled from the training data and a constant learning rate of 0.01 is applied without any annealing schedule for any of the training runs.

Since the goal of this experiment is the observation of performance of the optimization algorithms in general, no sophisticated enhancements to the training procedure are applied. That is, also no early stopping criterion is implemented and the performance of the model on the validation data has no impact on training. The experiment is conducted on a Linux

[3] Mixed National Institute of Standards and Technology database

[4] The recompiled version of the MNIST data used here is available at `http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz`

Ubuntu 16.10 machine equipped with 8 GB of RAM and an Intel i5 quad-core processor running at 2.60 GHz.

For training the network, the mean NLL of the correct classes is used as objective function. The network then is optimized by applying the algorithms Hogwild!, AsynchDA and AsynchAdaGrad as explained in Section 4.2. Each of these algorithms are run four times on 1, 2, 3 and 4 concurrent processes.

Since the training is performed on a machine with 4 CPU cores, it is to note that the underlying matrix multiplications are by default parallelized on CPUs as well. In this setting, therefore, race conditions between the two levels of parallelization are introduced and resolved in an unpredictable manner. To avoid distorted results, parallelization of the underlying computations is disabled. Data for training runs where parallelization of these calculations is allowed can be found on the disc attached to the printed version of this thesis.

For comparison reasons, another training is run with a native implementation of Sgd. This version of the sequential algorithm is far from optimal and additionally slowed down by measuring its performance on the different data sets throughout training. Nevertheless, the results below show that in this rather small setting, this basic version of Sgd still performs better than Hogwild! and AsynchDA.

5.3 Results and Discussion

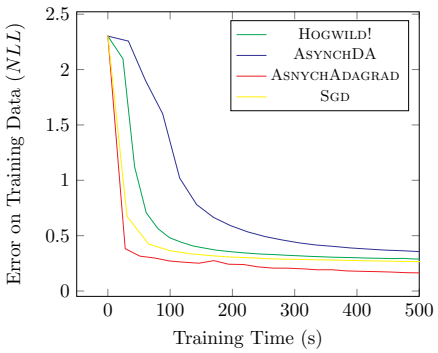
The performance of the algorithms is measured in terms of accuracy and error (mean NLL) on the different data sets. Accuracy means the percentage of correctly classified images in a given set. Figure 5.1 depicts the training process by plotting the error on the whole training data over time for the different algorithms and different degrees of parallelism. Figure 5.2 shows the accuracies on the test data set plotted over time for the same training runs. A CD is attached to the printed version of this thesis, containing additional numbers and figures.

All tested algorithms perform well and produce decent classification results within a few training iterations and in a small amount of time. AsynchAdaGrad shows the fastest rate of convergence.

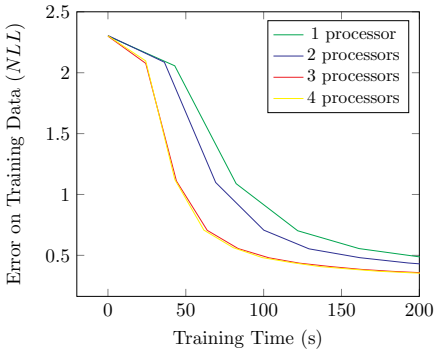
The number of concurrent processes has the most effect on the speed of each algorithm when increased up to 3. Figure 5.3 shows that the number of updates applied per second does not increase considerably when the number of concurrent processors is increased from 3 to 4. In this precise setting the gradients at each update can be calculated rather quickly and the message passing overhead between the processes rapidly outweighs the benefits of parallelism. This corresponds to the findings of Langford et al. (2009) that more simple problems implicating minor computations are more difficult to speed up. With growing complexity of the problem, the effect of parallelism becomes more visible. This is promising for the experiment conducted on optimizing an NMT system and generally confirmed in Section 6.

It is remarkable that AsynchAdaGrad reaches good values very fast and then only improves marginally. In fact, the speed of the algorithm in the first few steps is not even captured in the measuring interval of 500 updates (see Figures 5.1d and 5.2d).

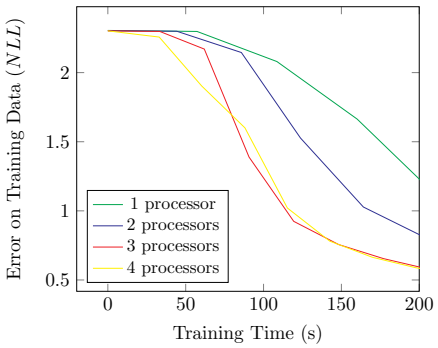
Figures 5.1a and 5.2a show that serial Sgd achieves near the same performance as the AsynchAdaGrad algorithm and performs even better than the Hogwild! algorithm where both of the asynchronous algorithms are run in parallel on 4 concurrent processors. Taking into account that the update schemes of Sgd and Hogwild! are the same, this



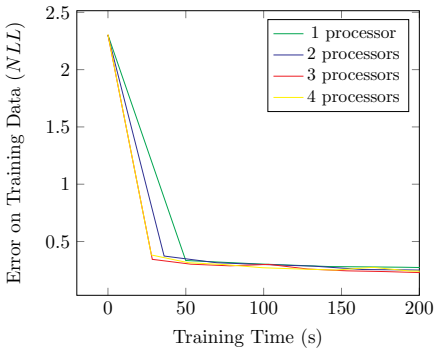
(a) Comparison of different optimization algorithms in terms of training error. Each asynchronous algorithm is run on 4 concurrent processors.



(b) Decrease of training error of the Hogwild! algorithm over time run with different numbers of concurrent processors



(c) Decrease of training error of the AsynchDA algorithm over time run with different numbers of concurrent processors



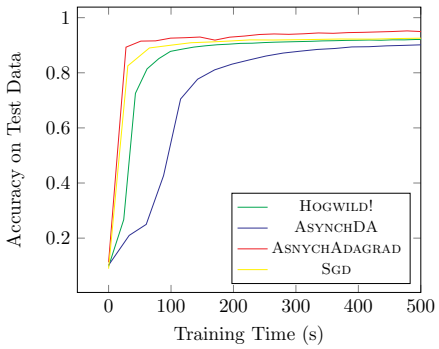
(d) Decrease of training error of the AsynchAdaGrad algorithm over time run with different numbers of concurrent processors. The measurement is not fine-grained enough to better capture intermediate improvements within the first few seconds of training.

Figure 5.1: Progress of training of asynchronous algorithms for classification of MNIST data over time. The asynchronous algorithms are run in parallel on CPU cores.

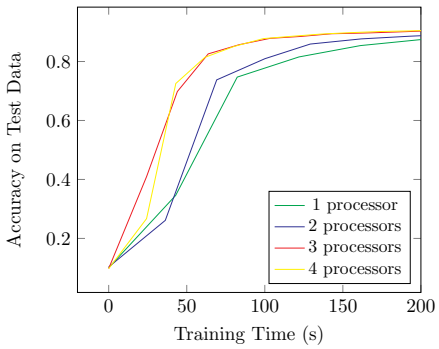
suggests that the effect of parallel computing in this setting is annulled by parallelization overhead.

A peculiarity observed during supplementary tests not reported here is that AsynchDA stagnated with low performance when applied to deeper models. Also, AsynchAdaGrad has shown worse performance on optimizing models with more than one hidden layer. While worth to note, the explanation of this behavior requires further elaboration which is out of the scope of this work.

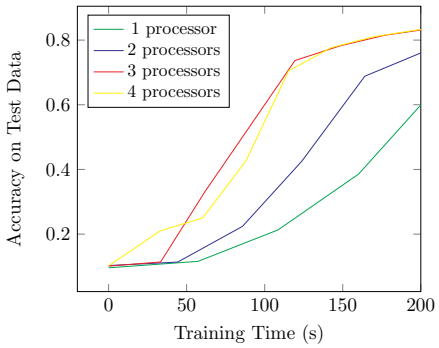
Allowing parallelization of the underlying matrix computations on all cores yields slightly better results for all algorithms. This is rather unsurprising as the usage of all available cores is also enabled even when the higher level parallelization is restricted to fewer processors. However, this phenomenon further encourages the use of GPU clusters where the low level calculations are parallelized on even more cores on each GPU while race conditions are avoided as parallelization of the optimization algorithm itself is kept separate on the different GPUs.



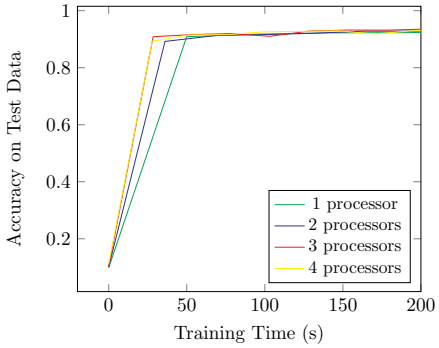
(a) Comparison of the algorithms in terms of accuracy on a test data set. Each asynchronous algorithm is run on 4 concurrent processors.



(b) Increase of accuracy on the test dataset for the Hogwild! algorithm over time run with different numbers of concurrent processors



(c) Increase of accuracy on the test dataset for the AsynchDA algorithm over time run with different numbers of concurrent processors



(d) Increase of accuracy on the test dataset for the AsynchAdaGrad algorithm over time run with different numbers of concurrent processors. The measurement is not fine-grained enough to better capture intermediate improvements within the first few seconds of training.

Figure 5.2: Increase of accuracies on test data of the MNIST dataset over time. The asynchronous algorithms are run in parallel on CPU cores.

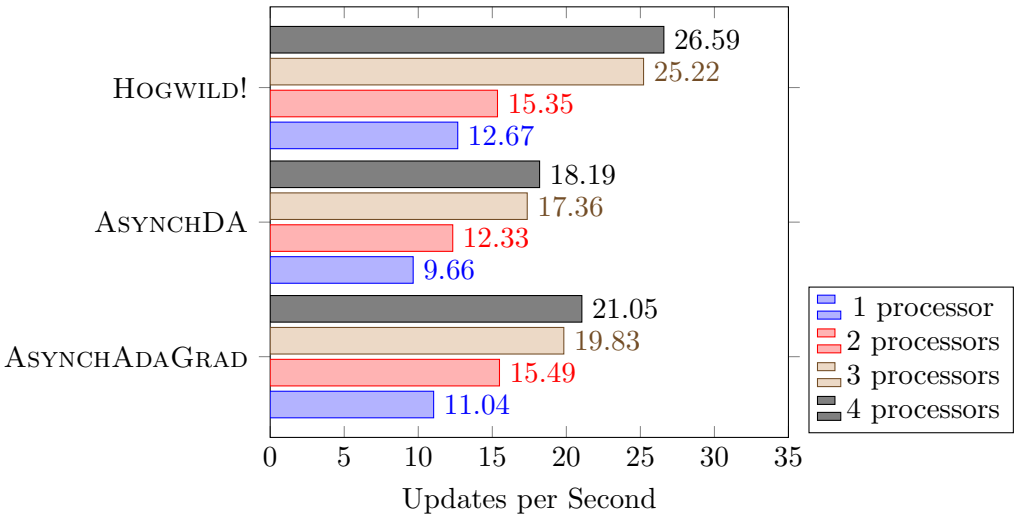


Figure 5.3: Updates per second of the asynchronous algorithms parallelized on different numbers of concurrent processors

5.4 Conclusion

The preliminary experiment using the asynchronous optimization algorithms Hogwild!, AsynchDA and AsynchAdaGrad to train a NN model for classification of the MNIST data set showed that these algorithms can be applied to such a problem. The performance of the algorithms for this particular task, however, is not outstanding and a standard implementation of Sgd yields comparable results. The effect of parallelization can be observed with regard to the number of updates applied per second which results in a slight improvement in classification performance. Nevertheless, in this setting these benefits are predominated by parallelization overhead relatively fast.

6 Parallel Training of Neural Machine Translation Models

This section will describe the experiment carried out for this work in detail. An NMT model is trained for an English-to-French translation task using different optimization algorithms. Apart from the sequential algorithms listed in Section 4.1, the focus is on the asynchronous optimization algorithms Hogwild!, AsynchDA and AsynchAdaGrad (Section 4.2).

After commencing with an outline of implementation details and the data base, the experiment settings are described. Before depicting the achieved results, the hardware specification of the machines on which the experiment is run is given.

6.1 Implementation

The NMT system for this work is implemented using the `theano` library (Theano Development Team, 2016) in Python 2.7. `theano` offers a convenient way of implementing the operations commonly used for NNs in an efficient computing graph structure. An additional benefit of `theano` is the support of symbolic differentiation. This is especially useful for gradient based optimization algorithms. Swapping computation to GPUs which are very well suited for the matrix computations NNs are based on is also very easy.

The codes for training an NMT model build strongly on code released for the DL4MT tutorial[5]. The main modifications applied to this code base are concerned with adding the parallelizable optimization algorithms. Parallelization is implemented using Pythons `multiprocessing` module.

The general structure of the parallel program is as follows. A main process sets up the initial parameters of the model and starts subprocesses which each generates and compiles a local copy of the computational graph defining the operations carried out during training. This is necessary due to `theano` functions not being thread safe. The main process then sends batches of training data and optionally receives information about the training process from the subprocesses. Dependent on the applied algorithm, the shared parameters (Hogwild!) or shared gradient sums (AsynchDA, AsynchAdaGrad) are updated by the individual processes with or without locking. Saving intermediate models for later inspection is handled by the main process which can also be set to evaluate the model in certain intervals possibly triggering an early stopping of training. Models are saved as `numpy npz` files storing the weights and biases of the different layers of the model while also comprising information about the training process.

Since the parameters of the model in AsynchDA and AsynchAdaGrad are calculated on each processor individually, they are reported back to the main process after each iteration. The parameters then are store according to the saving interval. This does not ensure that the best parameters currently computed by any of the processors are stored but rather a rough estimate thereof. However, in the experiments carried out for this work, this strategy shows reasonable behavior.

After reaching the limit of maximal updates or epochs, the latest model and the model evaluated as performing best on the validation data are stored to disc. If an possible early stopping criterion is met, the training terminates before reaching one of the set limits. When training is completed, the main process sends a special stopping signal to all subprocesses and waits for their termination before the whole program eventually terminates.

[5] `https://github.com/nyu-dl/dl4mt-tutorial`

While tested on multiple CPU cores, the implementation supports execution on GPU clusters as well.

For evaluation of the models, a separate script calculates the error (mean NLL) on a given data set for all models in a specified directory. The results are written to `csv` files and thus can easily plotted with other data processing tools. The `csv` files obtained throughout the experiment can be found on the CD attached to the printed version of this thesis.

Another script can be used to translate a file with sentences of the source language into the target language with a previously trained model. The source sentences are transformed into a list of vector representations of the words and processed by the model. The resulting probability distributions are then sampled to target words one by one as described in Section 3.7.

It is to note that each of the scripts for training, evaluation and translation require vocabularies for the source and target language in form of Python `pickle` files containing a mapping of words to their indexes as `dicts`. The original DL4MT codes include a script to generate such files from tokenized text files.

Due to some specifics of the Python programming language, the implementation of the HOGWILD! algorithm slightly differs from the one described in Recht et al. (2011). A complex data structure based on `C` data types is implemented. In contrast to the requirement of the original algorithm to update the atomic parameters, this data structure only allows updating the weights and biases of a layer in the network all at once. Nevertheless, the implemented version of HOGWILD! showed reasonable performance so that this peculiarity can be neglected.

It is also to note that the implementation of RMSPROP is altered by the authors of the DL4MT code and does not exactly match the definition of the algorithm by Tieleman and Hinton (2012) which is described in Section 4.1.3.

Another peculiarity is that the implementation of standard SGD is also subject to minor adjustments in order to fit the interface of other optimization algorithms in the original DL4MT codes.

The implementations of ASYNCHDA and ASYNCHADAGRAD also differ from their description in Duchi et al. (2013). In the first iteration, the sums of gradients are clearly zero for each parameter. Solving the inner optimization problem (Equations 39 and 41) clearly yields parameters with all entries set to zero. Thus further updates become meaningless. To workaround this, the first gradients are computed with the initial random parameters and added to the sums accordingly.

All source codes are available online at `https://github.com/valentindey/parallel-nmt` and on the CD attached to the printed version of this thesis.

6.2 Data

The data used for the experiments described here consists of parallel texts in English and French.

Training data consisted of a subset of the *europarl* corpus[6]. The original corpus contains more than 2 million translated sentences for the two languages, summing up to almost 56 million words in English and almost 62 million words in French, totaling to about 118 million words. For this work, the first 10.000 sentence pairs of the corpus were extracted, which contain 283.552 words in English and 308.179 words in French, summing up to 591.731 words in total. While this smaller amount of data is clearly not enough for

[6] The europarl corpus is available online at `http://www.statmt.org/europarl/v7/fr-en.tgz`

training decent translation models, it is sufficient to measure the performance of different optimization algorithms used for the fitting of such models. For training itself, the amount of data used is not much of an issue since the data is streamed from the files in a buffered way. Using more data is thus feasible without any further ado.

The training progress is also measured on data held out from training. This data consists of 3.003 sentence pairs from news articles in English and French.[7] The sentences sum up to 159.408 words, 74.800 in English and 84.608 in French.

All texts were tokenized with a tokenizing script supplied as utility to the SMT system *moses*.[8] The word counts above refer to token counts, that is, all separated punctuation and fragments like *l'* in French increase this count as well. No further preprocessing is applied to the data.

6.3 Model and Training Parameters

For training the models, the default settings from the original DL4MT code are maintained. That is, no regularization terms are added to the cost function and no gradient clipping is applied. A last *tanh* layer generating word probabilities conditioned on the context in the target language is subject to dropout. Batches of size 16 are processed in each iteration of the algorithms and a constant learning rate of 0.0001 is applied without any decay. There is no early stopping criterion defined as validation during training is disabled for this experiment.

Training is restricted by only considering sentences of maximum length of 100 tokens. This boundary applies to 80 sentence pairs in total (35 English sentences and 72 French sentences are affected) reducing the training data set size to 9.920 samples.

Intermediate models are saved in intervals ranging from 25 updates to 250 updates. The intervals are adjusted to fit the times needed for an update by the tested algorithm. However, the shortest saving intervals of 25 updates in some cases fail to closely capture very fast improvements of an algorithm in the beginning phase of training.

Preceded by an embedding layer of size 128, the encoder is realized as an RNN with GRUs of dimension 512. The following decoder has the same size and additionally applies the attention mechanism explained in Section 3.6. Another embedding layer of size 128 is appended from which the final probability distributions for the target words are generated.

For both languages, the 30.000 most frequent words are considered, replacing all other words with a special *UNK* token. The end of a sentence is marked with the *EOS* token.

Each training run is executed for about 20 to 24 hours and then halted as this duration is enough for a measurement of optimization performance.

In contrast to the preliminary experiment, parallelization of matrix multiplications on CPUs is allowed here as the used machines supply enough remaining threads (see the following section). In consequence, the findings are more comparable to results achieved on GPU clusters.

6.4 Hardware

All training runs with the different optimization algorithms were executed on two machines with the same hardware specifications. These machines operating on Linux Ubuntu 16.04

[7] The news article data set is available at `http://matrix.statmt.org/test_sets/newstest2011.tgz`

[8] The tokenizing script can be found in the moses repository at `https://github.com/moses-smt/mosesdecoder`

are equipped with 144 GB of RAM and 2 Intel Xeon X5650 CPUs running at 2.67 GHz. Each of these CPUs has 6 cores supporting hyper-threading, resulting in 24 threads in total.

6.5 Results

While hardly noticeable, the results listed below are marginally delayed. After the parameters of a model are initialized, the compilation of the computational graph in `theano` takes relatively long. Compiling several of these graphs for each processor when the training is run in parallel does not affect this duration any further. The resulting delay from this effect is treated as belonging to the training time of a model. For the comparatively much longer durations of training, this amount of time is negligible anyways.

The training of NMT models with the sequential optimization algorithms listed in Section 4.1 yields reasonable results. Except for SGD and ADAGRAD, the serial algorithms show capability to reduce the error calculated on the training and test data considerably, following a similar pattern (see Figure 6.1). Regarding the performance of the algorithms, it is striking that after reaching an error value of around 200 very quickly, the decrease stagnates and oscillates around this value. In the beginning phase, ADAGRAD shows matching performance with SGD but after a few updates, the additional information it takes into account is reflected by its faster decrease of error.

After the first big drop in error, the model trained with ADAM experiences slight increase of error again. This error decreases again to a similar value as achieved with RMSPROP and ADADELTA after about 14 hours of training. This novel decrease is observable especially with regard to the error when measured on the training data set, where the corresponding model shows a new drop in error.

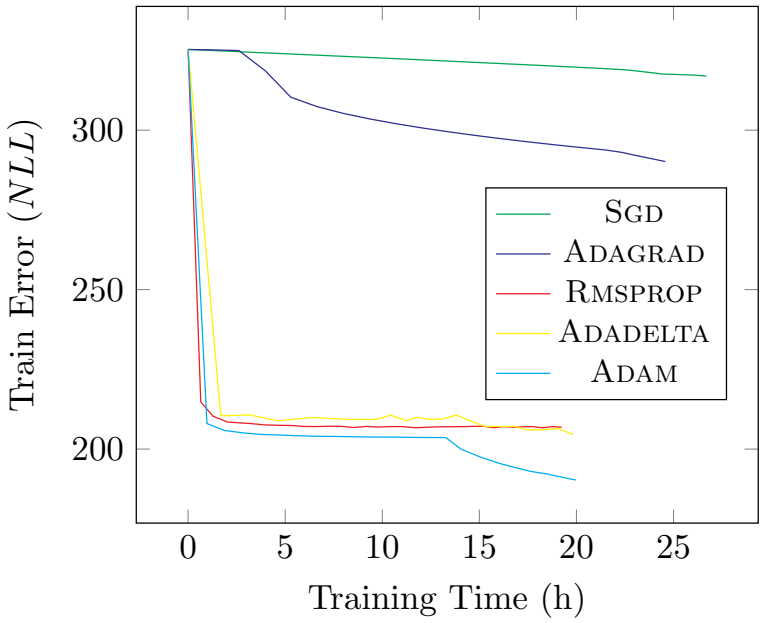
The general picture of performance measured over time is maintained when regarding the improvement per update. However, when comparing the two measuring forms (Figures 6.1b and 6.1c), it is visible that the updates for some of the algorithms take longer. This is notable, especially in a big drop in error after a few updates where the curves displaying development for ADADELTA, ADAM and RMSPROP over time are divided and overlap when plotted per update. Within the roughly 20 hours of training, the modified version of RMSPROP is able to perform measurably more updates.

Figure 6.2 shows the number of updates per hour performed by the individual algorithms. In Figure 6.2a it is obvious that increasing the number of concurrent processors to parallelize the asynchronous algorithms on, also increases the number of updates applied.

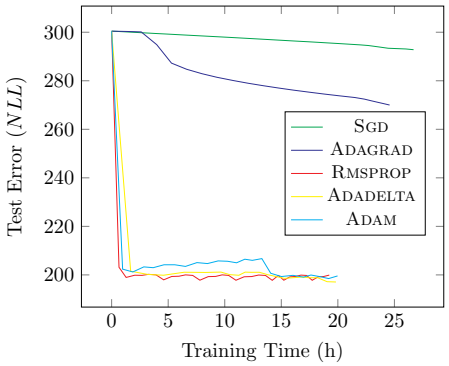
Considering the asynchronous algorithms, the speedup by increasing the number of concurrent processes is more marked than in the preliminary experiment shown and evaluated in Section 5.

The performance of the HOGWILD! algorithm is shown in Figure 6.3. When the number of concurrent processes is increased, the speedup of training is very well observable. However, the higher the degree of parallelism, the lower the improvement in terms of speedup that is achieved. This can be attributed to the increasing overhead by message passing between the main and the increasing number of subprocesses. Figure 6.3c and Figure 6.2a reveal that the convergence of the algorithm is the same across different degrees of parallelism when regarding the number of performed updates. This finding corresponds to the theory of Recht et al. (2011) and allowing possible overwriting of the shared parameters according to the lock-free update policy of HOGWILD! does not widely affect the course of training.

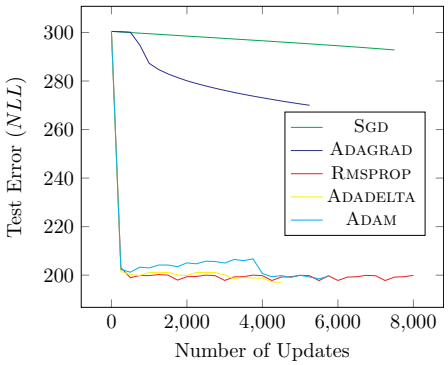
For ASYNCHDA and ASYNCHADAGRAD, the picture looks somewhat different. While these algorithms better the models as shown in Figure 6.4, this improvement is very slight



(a) Decrease of error on the training data over time for training an NMT model with sequential algorithms

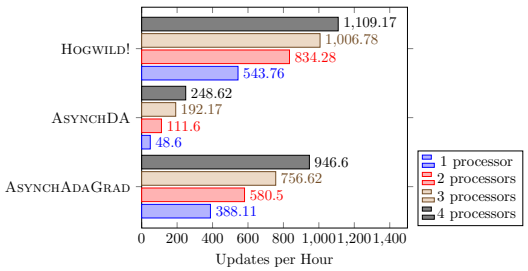


(b) Decrease of error on the test data over time for training an NMT model with sequential algorithms

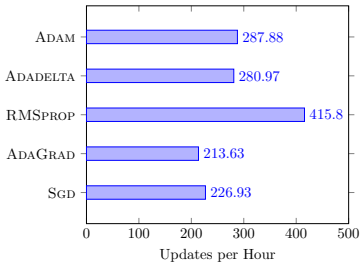


(c) Decrease of error on the test data per 250 updates for training an NMT model with sequential algorithms

Figure 6.1: Decrease of error during training of an NMT model with different sequential algorithms. The training is executed on CPU cores.

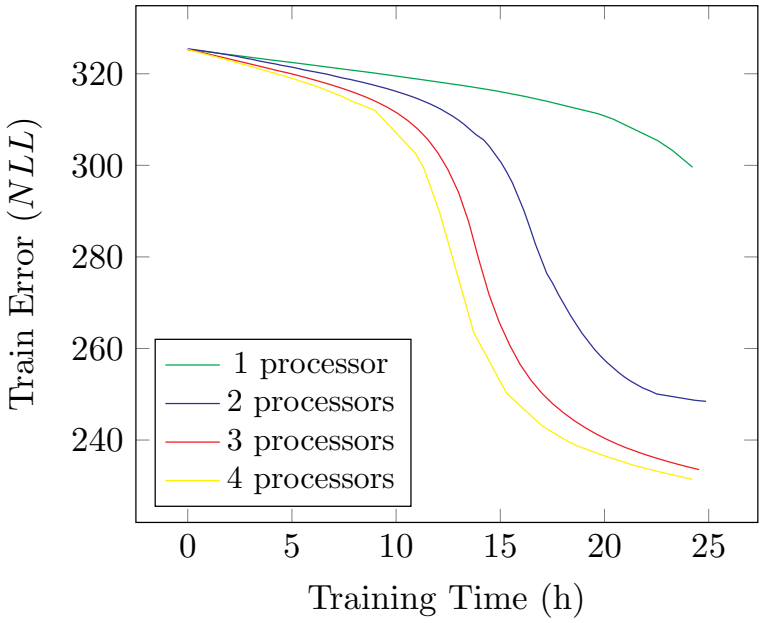


(a) Number of updates performed by the asynchronous optimization algorithms parallelized on different numbers of concurrent processors

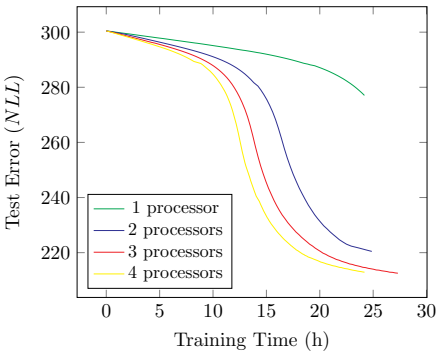


(b) Number of updates performed by the sequential optimization algorithms

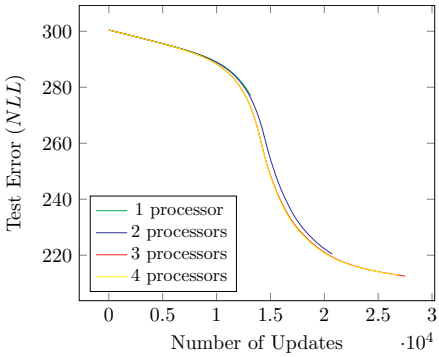
Figure 6.2: Number of updates performed per hour by the different algorithms



(a) Decrease of error on the training data over time for training an NMT model with the Hogwild! algorithm parallelized on different numbers of processors

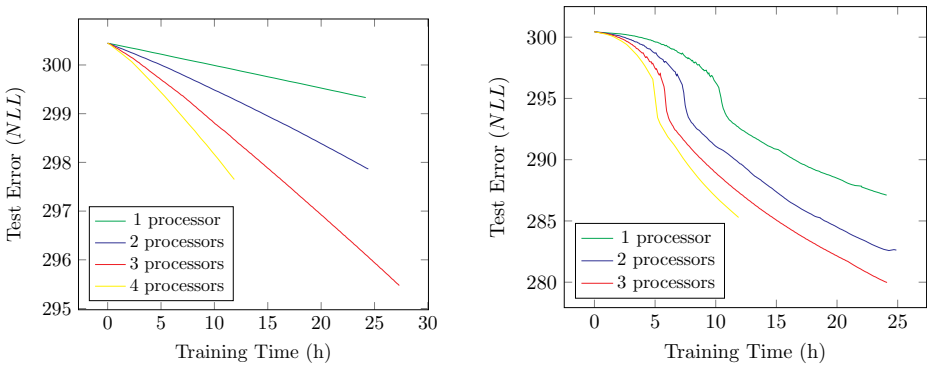


(b) Decrease of error on the test data plotted over time for training an NMT model with the Hogwild! algorithm parallelized on different numbers of processors



(c) Decrease of error on the test data per updates for training an NMT model with the Hogwild! algorithm parallelized on different numbers of processors

Figure 6.3: Decrease of error during training of an NMT model with the Hogwild! algorithm parallelized on different numbers of processors. The training is executed on CPU cores.



(a) Decrease of error on the test data over time for training an NMT model with the AsynchDA algorithm parallelized on different numbers of processors

(b) Decrease of error on the test data per updates for training an NMT model with the AsynchAdaGrad algorithm parallelized on different numbers of processors

Figure 6.4: Decrease of error during training of an NMT model with the AsynchDA and AsynchAdaGrad algorithm parallelized on different numbers of processes. The training is executed on CPU cores.

and compared to the performance of other algorithms including Hogwild!. AsynchDA and AsynchAdaGrad are not suited for optimization of NMT models.

Unfortunately the training for both algorithms parallelized on 4 CPU cores was aborted due to some unexplainable reason. Resuming the training from the last saved parameters is not feasible as the information about past gradients is missing.

To better set the performance of the asynchronous algorithms in context, Figure 6.5 shows their improvement in contrast to the performance of two of the sequential algorithms. The numbers reported in this figure are achieved with models trained by the asynchronous algorithms parallelized on 4 cores as they achieve the best results with this setting (reported above). It is also the highest degree if parallelism tested in this work and a common number of cores available in machines with multiple CPU cores. To not bloat the graphic, for the sequential algorithms only Sgd and Adadelta are shown. This choice is based on the first being the canonical gradient based optimization algorithm and the later being an adopted version thereof frequently used for training NMT models (Bahdanau et al., 2014; Cho et al., 2014b,a). The values for AsynchDA and AsynchAdaGrad are only reported for about 12 hours of training due to the error mentioned before. However, the general tendency of the corresponding trajectories is discernible.

While AsynchDA and AsynchAdaGrad improve the model only slowly, yielding relatively high cost values throughout the whole duration of training, AsynchAdaGrad run on 4 concurrent processors slightly improves the results achieved with standard Sgd. Among the asynchronous algorithms, Hogwild! shows the best performance, lowering the error value considerably in a steep decrease after about 15 hours of training. However, this drop is bottoming out again very fast and the curve depicting its progress approximates a mean NLL value of about 220. The best of the compared algorithms is Adadelta with almost immediately achieving a mean NLL as low as 200. Recalling the results presented earlier in this section, the same holds for Adam and the modified version of RMSprop.

Since the update schedules of Sgd and Hogwild! are the same, running the latter on only one processor results in almost identical procedures. Only the parallelization overhead introduced by the data sent to the process running the updates in Hogwild! makes this algorithm theoretically less performant. However, the results show that Sgd performs worse than Hogwild! run on one processor. This can only be attributed to the

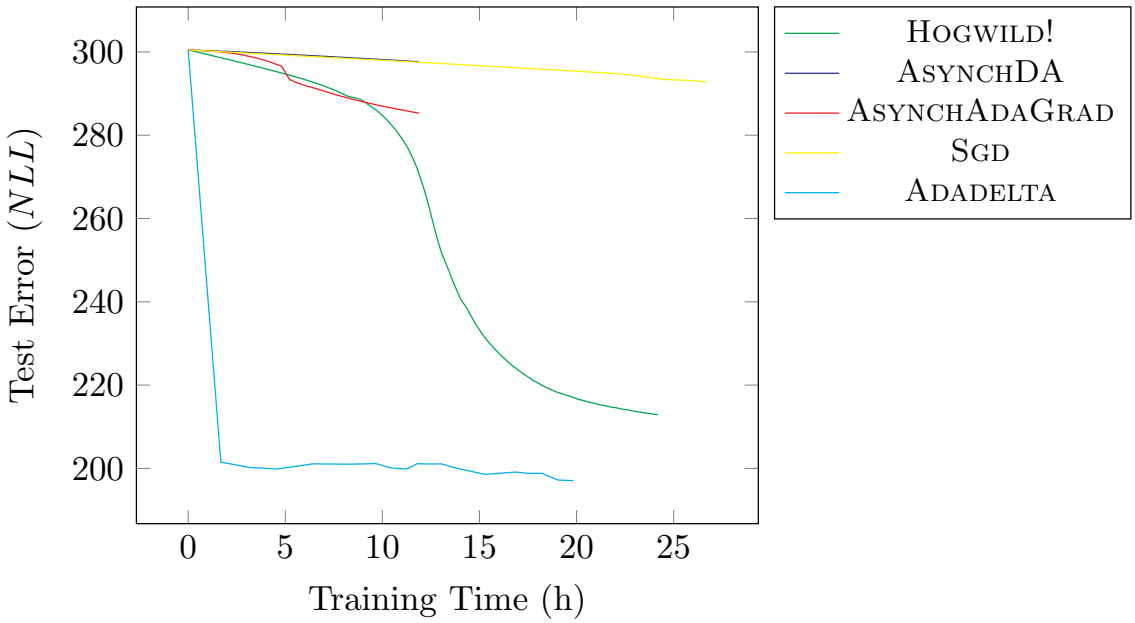


Figure 6.5: Decrease of error during training of an NMT model with different sequential and asynchronous algorithms. The asynchronous algorithms are parallelized on 4 concurrent processes. The training is executed on CPU cores.

somewhat inefficient implementation of SGD in the DL4MT code as mentioned before.

The results shown in this section reveal that the rather advanced sequential algorithms tested need less updates to improve the model in terms of training and test error. While the asynchronous algorithms HOGWILD!, ASYNCHDA and ASYNCHADAGRAD apply updates to the parameters at a faster pace, these updates do not better the model comparably. This is mainly due to the advanced sequential algorithms incorporating more information for the calculation of their updates by taking information about previous gradients and updates into account. While ASYNCHDA and ASYNCHADAGRAD also rely on the previously calculated gradients, they are inherently not suited for the task of optimizing NMT models. However, this statement is made restrainedly as the amount of data used in this experiment may be unfavorable towards these algorithms.

6.6 Conclusion

In this section the outcomes of training an NMT model with different optimization algorithms are listed. Characteristics of the implemented system and optimization algorithms are listed, pointing out specifics concerning the latter. A description of the datasets used for training and testing this system is given. After setting out the hyperparameters affecting the model and the training process, the used hardware is outlined and the results of different training runs are displayed. The asynchronous algorithms are tested with parallelization on up to 4 processors. The performance of the training algorithms is measured in terms of error calculated on the training and test data sets. With respect to training time and the number of performed updates, it has been shown that elaborate sequential algorithms attain superior convergence properties. While increasing the number of concurrent processes to run on yields speedups for all tested asynchronous algorithms, only HOGWILD! shows optimization performance that can keep pace with the sophisticated sequential algorithms. The following section will elaborate on these results.

7 Discussion

This work showed that the asynchronous optimization algorithms Hogwild!, AsynchDA and AsynchAdaGrad can be applied to train NNs of different complexity to a certain extend. However, it is shown that the two latter methods are rather inadvisable for training NMT models or other more complex NN models. However, the small amount of data used for training in this experiment may be unfavorable to these algorithms. Increasing the training data possibly could improve their optimization performance. This is mainly due to them being designed relatively strictly towards solving convex problems. In contrast, Hogwild! achieves very reasonable results, surpassing standard Sgd and serial AdaGrad when run on as few as 2 concurrent processes. This performance is only outdone by the more advanced sequential algorithms Adadelta, Adam and a modified version of RMSprop. These algorithms show the best convergence properties in the experiment conducted for this work, with the adaption of RMSprop marginally leading the way. The major difference between these algorithms only is measurable in terms of time the updates need to be applied. In this regard, the adapted version of RMSprop also showed the fastest runtime.

Considering the updates performed per hour as shown in Figure 6.2, it is striking that the sequential algorithms perform clearly less updates in general. With this in mind, the superior optimization performance of these algorithms shows that the updates utilize the additional information they take into account effectively.

The observation of error stagnating at a relatively high level con most certainly be attributed to the small size of training data in which only a few constant regularities of the languages considered are observable. This observation can be made throughout all experiments and thus suggests that the small amount of data is too diverse to really be suited for training decent NMT models. It is also apprehended that the few data restricts validity of the findings of this work.

The peculiar trajectory of training error throughout training with Adam (Figure 6.1) shows that there is potential for improvement even with the relatively small model and the little training data. However, a quick test with the model attaining the lowest training error (mean NLL value of 190.30) and relatively low test error (mean NLL value of 199.57) showed no acceptable translations. The translations obtained with this model are mere concatenations of incoherent words and are not worth listing in the preceding section.

Comparing the two separate experiments reveals that smaller problems like training an NN for classification of the MNIST data does not undergo the same improvements from parallelization a more complex problem like NMT does, confirming the findings of Langford et al. (2009). While increasing the number of concurrent processes yields further speedup for each increase up to 4 and possibly more processors in the case of NMT, for the MLP tested on the MNIST classification, there is no significant improvement when increasing the number of concurrent processors from 3 to 4 for any of the tested algorithms. It is to note that the two experiments were run on different hardware and, thus, this statement may not be universally valid. However, for the more basic problem, the parallelization overhead slows down training altogether wherefore a native sequential optimization algorithm is to prefer in this setting. This can be explained by the relatively quick computation of gradients in the smaller setting getting predominated by read-write overhead between processes very quickly.

Another interesting insight is revealed by comparing AdaGrad to its asynchronous counterpart. While both algorithms are not very efficient in terms of optimizing an NMT model, the sequential version achieves considerably better results. Even when executed in parallel on 4 concurrent processes, the performance of AsynchAdaGrad is inferior.

As this work is only concerned with speedups achievable in training, measuring performance of the algorithms solely based on the errors obtained with the models throughout training comes natural. Further quantitative and qualitative evaluation, for example measuring BLEU scores the models achieve, is left aside. Nevertheless, it is anticipated that there is a high correlation between improvements in terms of lower error and actual translation quality. Again, the small amount of data used for the experiment makes measuring performance more exactly rather difficult.

The most clearest statement that can be made is that AsynchDA is an optimization algorithm not suited for the task of training complex NN models. The related algorithm AsynchAdaGrad also seems to suffer from similar drawbacks when applied to training of complex models. However, this effect may be counteracted by increasing the training data.

It should also be noted that the programming language Python is not the best choice for implementing parallel programs. While it does support threads, actual parallelism is prevented by the Global Interpreter Lock (GIL) allowing Python byte code execution of only one thread at a time (Python Software Foundation, 2015). The `multiprocessing` module constitutes a workaround to this problem by offering a way to maintain separate forks of the original process for each of the run processes. However, this comes at the cost of relatively expensive data copying overhead. Data has to be serialized and deserialized whenever passed between processes. Despite this weakness, the experiment for this work was implemented in Python due to the fact that the most commonly used and advanced deep learning libraries and also the modified code base are available for Python. The sophisticated data structure mentioned in Section 6.1 still enabled low level shared memory between processes.

8 Future Work

A major shortcoming of this work is that the implementation could only be tested on machines with multiple CPU cores. Due to their great performance for the applied optimization algorithms, GPUs are widely used to train NNs and state-of-the-art NMT systems (see for example Cho et al., 2014b). Since the implementations for this work are also executable on multiple GPUs, testing the implementation in an appropriate setting is an important and easy next step. However, GPU clusters are relatively expensive and not a setting available to all researchers. Therefore, this work already made an important contribution in that it compared different optimization algorithms run on machines with multiple CPU cores.

The decreasing performance gain when increasing the degree of parallelism visible in Figure 6.3b demands for further investigation of this phenomenon. Knowing the limits of parallelization is crucial for the meaningfulness of further studies in the field.

To increase the significance of the results and broaden the context of this work, additional asynchronous optimization algorithms would need to be tested. An interesting approach would be the implementation of several algorithms within the framework presented in Reddi et al. (2015).

A comparison of the data parallel algorithms tested in this work to model parallel approaches like implemented by Sutskever et al. (2014) could also improve the significance of this work by revealing which of the two approaches is more promising for further research and better suited for training NMT models.

This work is only concerned with measuring speed of training not taking into account actual translation performance of the resulting models. Extending the training data set to obtain better models and evaluating them in terms of BLEU score or examining qualitative results could further elaborate the findings and provide additional insights about training performance. In the same tenor, the size and depth of the model as well as the training duration could be increased to improve the quality of the resulting models even more and, thus, make the outcomes more meaningful.

As the results have shown, especially the Hogwild! algorithm has proven to be suitable for optimization of NNs. Testing this algorithm for other applications where DNNs are used could reveal further insights for the performance of this algorithm. Furthermore, the implementation at hand used for NMT training could easily be used for and tested on other sequence-to-sequence learning tasks.

Motivated by the protruding results of sophisticated sequential algorithms in comparison to the Hogwild! algorithm which still performs reasonably well, a novel optimization algorithm could be developed. While taking into account more information than just the gradients computed for the current batch, this algorithm would incorporate information gathered in earlier updates while altering the parameters in the fashion of Hogwild! combining the strengths of both approaches.

Testing the current state of the model on validation data during training to possibly trigger an early stopping is not yet supported flawlessly. Large amounts of validation data could introduce unpredictable behavior when validation is executed within the main process. This problem was tackled during development of the system but was not exhaustively tested as it was not important for the work in hand. As closing remark, it should be noted that eliminating this problem completely is a necessary step for using the implementations of this work in productive environments.

References

Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016). Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*.

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828.

Bojar, O., Chatterjee, R., Federmann, C., Haddow, B., Huck, M., Hokamp, C., Koehn, P., Logacheva, V., Monz, C., Negri, M., Post, M., Scarton, C., Specia, L., and Turchi, M. (2015). Findings of the 2015 workshop on statistical machine translation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 1–46, Lisbon, Portugal. Association for Computational Linguistics.

Brown, L. (2014). Accelerate machine learning with the cudnn deep neural network library. `https://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library`. [Online; accessed 2016-07-14].

Chan, W., Jaitly, N., Le, Q., and Vinyals, O. (2016). Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4960–4964. IEEE.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011). Convolutional neural network committees for handwritten character classification. In *2011 International Conference on Document Analysis and Recognition*, pages 1135–1139. IEEE.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

Defazio, A., Bach, F., and Lacoste-Julien, S. (2014). Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems*, pages 1646–1654.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Duchi, J., Jordan, M. I., and McMahan, B. (2013). Estimation, optimization, and parallelism when data is sparse. In *Advances in Neural Information Processing Systems*, pages 2832–2840.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

Johnson, R. and Zhang, T. (2013). Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709, Seattle, Washington, USA. Association for Computational Linguistics.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., Cowan, B., Shen, W., Moran, C., Zens, R., et al. (2007). Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics.

Konecnỳ, J. and Richtárik, P. (2013). Semi-stochastic gradient descent methods.

Langford, J., Smola, A., and Zinkevich, M. (2009). Slow learners are fast. *arXiv preprint arXiv:0911.0491*.

LeCun, Y., Cortes, C., and Burges, C. J. (1998). The mnist database of handwritten digits.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Interspeech*, volume 2, page 3.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pages 543–547.

Nesterov, Y. (2009). Primal-dual subgradient methods for convex problems. *Mathematical programming*, 120(1):221–259.

Olah, C. (2015). Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs`. [Online; accessed 2016-07-22].

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Python Software Foundation (2015). Global interpreter lock. `https://wiki.python.org/moin/GlobalInterpreterLock`. [Online; accessed 2016-07-29].

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701.

Reddi, S. J., Hefny, A., Sra, S., Póczos, B., and Smola, A. J. (2015). On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances in Neural Information Processing Systems*, pages 2629–2637.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. `http://sebastianruder.com/optimizing-gradient-descent`. [Online; accessed 2016-07-24].

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Schmidt, M., Roux, N. L., and Bach, F. (2013). Minimizing finite sums with the stochastic average gradient. *arXiv preprint arXiv:1309.2388*.

Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.

Schwenk, H. (2012). Continuous space translation models for phrase-based statistical machine translation. In *COLING (Posters)*, pages 1071–1080.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

Tieleman, T. and Hinton, G. E. (2012). Coursera: Neural networks for machine learning.

Weaver, W. (1949). Translation. [Memorandum].

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2(3):5.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

List of Tables

List of Figures

Acknowledgements

I want to thank **Dr. Alexander Fraser** and **Dr. Tsuyoshi Okita** for their advice and support throughout this project.

Another thanks goes to **Danny Kühner** for his proofreading and help with all kinds of questions concerning writing.

I also want to thank **Florian Pfingstag** and **Johannes Baiter** for their technical support, inspiring conversations and for being the great fellow students they are.

Last but not least, a very special thanks is due to **my whole family** and **Katja Königstein** for their constant support without which this work would not have been possible.

Content of the attached CD

On the CD attached to the printed version of this thesis, the source code for the conducted experiments can be found in the directory `parallel-nmt`. This source code is also available online at `https://github.com/valentindey/parallel-nmt`.

The directory `tex` on the disc contains the LaTeX-files for this thesis and the extended data obtained through the conducted experiments. Additional figures can be easily compiled with the corresponding LaTeX-files located in `tex/figures`.

All literature consulted for this work that is available in electronic form is found in the directory `tex/literature`.