Bachelorarbeit im Fach Informatik
Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 6
Prof. Dr.-Ing. H. Ney

Extension of the Attention Mechanism in Neural Machine Translation*

27. März 2018

vorgelegt von:
Christopher Jan-Steffen Brix
Matrikelnummer 343982

Gutachter:
Prof. Dr.-Ing. H. Ney
Prof. B. Leibe, Ph. D.

Betreuer:
M.Sc. P. Bahar

*Updated version. The submitted thesis is available at the RWTH Aachen.

Erklärung

Brix, Christopher Jan-Steffen 343982
Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit~~/Bachelorarbeit/ ~~Masterarbeit~~* mit dem Titel

Extension of the Attention Mechanism in Neural Machine Translation

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 27. März 2018

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Aachen, 27. März 2018

Ort. Datum Unterschrift

Abstract

Recently, machine translation (MT) has been significantly improved by the usage of neural networks (NNs). Neural machine translation (NMT) allows to read a source sentence in a given language and to output the translation in another. The most promising results are reported based on an encoder-decoder architecture with an additional attention mechanism. There, the encoder reads the source sentence and generates a set of source representations. The decoder outputs a sequence of variable length given the target history and a context vector. This context vector is determined by the attention layer and is dependent on the source representations and the target history. To this end, the attention layer selects the currently important positions in the input, therefore creating an alignment between source and target. A lot of research is put into different definitions of the attention layer.

In this bachelor thesis, we evaluate the impact of making the encoder depend on the decoder and therefore recomputing the encoding at every time step. Furthermore, we try to provide the attention layer with the knowledge of which source words it attended to at the current and at previous time steps. We use recurrent neural networks (RNNs) that can process sequences of arbitrary length to process the source representations and generate the context vector for the decoder. Because basic RNNs suffer from vanishing and exploding gradients, we use long short-term memory (LSTM) cells and gated recurrent units (GRUs). To further improve the model, we add an additional attention layer on top of the RNN to compute the context vector as a weighted sum. Besides that, we generalize the concept of RNN attention layers to multiple dimensions. We use two-dimensional LSTM (2DLSTM) to process both the source sentence and the target history at the same time. In this topology, we view the translation as a two-dimensional mapping of the source and target into a shared vector space. By allowing the 2DLSTM to update the source representations based on the current decoder state, we hope to provide the network with a deeper insight into the dependency between both sentences. We verify that it is beneficial to use one- and two-dimensional RNNs as the attention layer, even though they take significantly more time to train.

Furthermore, we propose a novel architecture that combines the encoder, attention layer and decoder within either one or two 2DLSTM layers. We evaluate its performance with respect to the source sentence length and show that it is able to slightly outperform a conventional encoder-decoder architecture with an attention layer.

Contents

Chapter 1

Introduction

With over 7,000 spoken languages in the world [Simons and Fennig, 2017], the ability to translate them can be very helpful. Unfortunately, manual translation is very labor intensive and therefore costly. If the language is rarely spoken, it might even be difficult to find a bilingual translator.

Machine translation (MT) tries to solve this issue by translating documents from the source to the target language without the need of human interaction. The required work is thereby no longer dominated by the size of the corpus that should be translated. Instead, the main issue becomes how to teach the computer to translate correctly. Once this is accomplished, the computer can use the learned knowledge to translate an unlimited amount of data.

In recent years, significant progress has been achieved using neural machine translation (NMT). Instead of supplying the computer with a fixed set of rules that specify the individual grammatic rules of each language, in NMT the computer learns based on a training set. This set consists of a large corpus with one half being considered the source, the other the target data.

The process of training an NMT system involves the estimation of millions of parameters. Its ultimate goal is to generalize the ability to translate beyond the training corpus to allow the translation of unseen sentences.

A translation process can typically be split into three stages. First, the source sentence must be transformed in a way that allows the network to extract the meaning the sentence conveys. Then, some words of the sentence are selected to be translated next, conditioned on the already generated target history. Finally, the next target word is determined.

In popular models, the second step is made for all source words independently. We present alternative neural network (NN) structures that allow to select source words for the next translation step only if specific other source position have been selected as well. To this end, we apply recurrent neural networks (RNNs) that have an internal state to the source sentence. We also extend this approach to two-dimensional LSTM (2DLSTM). Finally, we evaluate the performance of a network only consisting of a single 2DLSTM, thereby avoiding the need for an explicit encoder or decoder.

1.1 Related Work

NMT has recently emerged as a promising approach to MT. It has been shown to outperform the conventional statistical machine translation (SMT) systems. Sutskever et al. [2014] and Cho et al. [2014] have created the first NMT systems that use RNNs for an encoder-decoder network. There, one RNN is used to encode the whole source sentence and another RNN generates an output hypothesis.

With the invention of the attention mechanism by Bahdanau et al. [2014], the ability to translate long sentences has been greatly improved. It determines which source positions the decoder should attend to next. Hence, the NMT system is provided with an implicit alignment model.

Zhang et al. [2016] propose to instead use a one-dimensional (1D) gated recurrent unit (GRU). This GRU processes the source representations at every decoding step and internally constructs the context vector that is then used by the decoder. They report significant improvements over the vanilla attention based model. In [Zhang et al., 2017], the authors use a GRU to update the encoding of the source sentence at every decoding step.

Graves et al. [2007] introduce the concept of multi-dimensional RNNs. They show that a 2DLSTM trained for single digit handwriting recognition is more robust to deformations that are only applied to the test set than convolutional networks. Graves and Schmidhuber [2008] apply them to handwriting tasks containing complete words or even lines of text. There, they establish a new state of the art. Additionally, they provide the explicit formulas for the forward and backward pass of a multidimensional long short-term memory (LSTM).

For 2DLSTMs, Voigtlaender et al. [2016] describe optimizations that reduce the time complexity of computing an $n \times m$ 2DLSTM from $\mathcal{O}(nm)$ to $\mathcal{O}(n+m)$. They implement it for GPUs in the RWTH extensible training framework for universal recurrent neural networks (RETURNN) that is introduced in [Doetsch et al., 2016]. Due to the increased speed of the 2DLSTM computation that is gained both by their optimization and the GPU implementation, they are able to experiment with architectures consisting of deeper 2DLSTMs. Using such a deep multidimensional network, they outperform the state of the art on two test sets.

Li et al. [2016] use 2DLSTMs to solve automatic speech recognition (ASR) tasks by processing the data both across the time and the frequency dimension. Sainath and Li [2016] compare different ASR models, including one that uses 2DLSTMs.

Kalchbrenner et al. [2015] apply another modification of LSTMs to machine translation. They propose a grid LSTM that has n memory and state cells to communicate with neighboring cells across its n dimensions. Based on this grid LSTM, they re-encode the source sentence at every decoding step based on the target history.

1.2 Outline

This bachelor thesis starts with a general introduction to MT (see Chapter 2) that explains the history of MT and the differences between rule driven and data driven translation models. Chapter 2 then continues with an overview of SMT, including the generation and evaluation of hypotheses (see Section 2.2 and Section 2.3, respectively)

Chapter 3 provides all necessary information about artificial neural networks (ANNs), from their general structure (see Sections 3.1 and 3.2) to the more complicated RNNs (see Section 3.3). The training process, consisting of the choice of loss function, the initialization and finally the gradient computation and weight update, is explained in Section 3.4.

In Chapter 4, NMT is introduced. The structure of a sequence to sequence model using an encoder and a decoder is described in Section 4.1. Attention, a technique to greatly improve the translation quality, is addressed in Section 4.2.

Chapter 5 lists and explains different extensions of the attention mechanism, mainly focusing on the application of 1D (see Section 5.2) or two-dimensional (2D) (see Section 5.3) RNNs. A novel approach that eliminates the need for the currently used encoder-decoder structure is presented in Section 5.4.

In Chapter 6, the general setup of the performed experiments is explained. The experimental results of the proposed modifications are listed in Sections 6.3 to 6.6.

Finally, in Chapter 7, a summary of all findings is presented. Further avenues of work that arise from this bachelor thesis are listed in Section 7.2.

Chapter 2

Statistical Machine Translation (SMT)

The general task of machine translation (MT) is to translate a given text from one natural language into another using an automatic mechanism. To this end, many different techniques exist. They can be categorized as either rule-based or data-driven.

In a rule-based system, the computer is programmed with a dictionary, as well as a list of the grammatical rules in both the source and the target language. These rules are defined by expert linguists. Based on the grammar of the source language, the computer tries to determine the meaning of the source words. To this end, it performs classifications like verb and noun discrimination. Using a dictionary, it can then translate the words to the target language. Afterwards, the stored information about the target language's grammar is used to reorder and adapt the translated words.

As all actions happen due to a specific combination of defined rules, the translation process of such a model can be debugged relatively easily. Together with the ability to define additional rules to fix mistakes that have been identified, this could in theory lead to perfect translations. However, the amount of the necessary work to create complete dictionaries and sets of rules that are fine-grained enough to reflect all nuances of a given language is prohibitively high. As the model becomes more and more complex, it gets increasingly difficult to extend it with additional rules.

A data-driven approach, on the other hand, derives the rules from a set of exemplary translations, called *training set*. In MT, this set consists of a large amount of sentences that have been manually translated from the source to the target language. The idea behind this approach is to enable the computer to detect the rules behind the translations itself, removing the need to manually define them. A single source sentence can often be translated into multiple different sentences in the target language, that may differ in their wording and structure but convey the same meaning. Nevertheless, it is definitely possible to identify good and bad translations. Data-driven models apply statistical methods. There, sentences are scored based on how likely they are to be a correct translation.

In this work, we only address data-driven approaches in MT. The following section describes the fundamental equations of SMT. A comparison of rule-based and data-driven methods can be found in [Costa-Jussà et al., 2012]. It also describes the structure of a rule-based machine translation model.

2.1 Fundamental Equations

SMT is based on the assumption that the translation task can be represented by a maximization problem. Given a source sentence $f_1^J = f_1, \ldots, f_J$, where f_j is the jth word, a target sentence $\hat{e}_1^{\hat{I}} = \hat{e}_1, \ldots, \hat{e}_{\hat{I}}$, where $\hat{e}_i$ is the ith word, has to be found that is the most likely translation. This can be formalized as

$$\hat{e}_1^{\hat{I}} = \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \Pr(e_1^I | f_1^J) \right\} \tag{2.1}$$

where $\Pr(e_1^I | f_1^J)$ is the true probability of e_1^I being the translation of f_1^J and E is the set of all possible sentences in the target language. Because no perfect probability distribution $\Pr(e_1^I | f_1^J)$ is known and the number of possible sentences $e_1^I \in E$ increases exponentially with the sentence length I, $\hat{e}_1^{\hat{I}}$ can only be determined approximately.

Brown et al. [1993] found that the probability distribution $\Pr(e_1^I | f_1^J)$ gives far too much weight to sentences e_1^I that are not well formed, ie. that do not follow the grammatic rules of the target language. Using Bayes's theorem, described in [Bayes, 1763], and by ignoring terms that are independent of the target sequence, the formula can be rewritten as follows:

$$\hat{e}_1^{\hat{I}} = \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \Pr(e_1^I | f_1^J) \right\} \tag{2.2}$$

$$= \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \frac{\Pr(f_1^J | e_1^I) \cdot \Pr(e_1^I)}{\Pr(f_1^J)} \right\} \tag{2.3}$$

$$= \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \Pr(f_1^J | e_1^I) \cdot \Pr(e_1^I) \right\} \tag{2.4}$$

As a decomposition of the target sequence posterior probability, the target sequence prior probability, called *language model* (LM) $\Pr(e_1^I)$, and a generative class-conditional distribution, called the *inverted translation model* $\Pr(f_1^J | e_1^I)$ are defined. $\Pr(e_1^I)$ indicates how likely the sentence e_1^I is to be used in the target language and measures whether e_1^I is a grammatically correct sentence.

The LM as described by Brown et al. [1990] uses the chain rule of probability theory to separate the computation of $\Pr(e_1^I)$ to multiple steps:

$$\Pr(e_1^I) = \prod_{i=1}^{I} \Pr(e_i|e_0^{i-1}) \tag{2.5}$$

here, e_0 is a special token that indicates the beginning of the sentence. e_0^{i-1} are called the *history* of e_i.

To simplify this estimation, one can assume that the probability of the occurrence of a word is not defined by the whole history, but only by the most recent part of it. Using the Markov assumption, the dependencies are limited to the previous $n-1$ consecutive words. This simplification yields:

$$\Pr(e_1^I) = \prod_{i=1}^{I} \Pr(e_i|e_{i-n+1}^{i-1}) \tag{2.6}$$

Due to the fact that it only depends on n consecutive words, this is called an *n-gram* LM. It is commonly used in practice and can be learned from mono-lingual corpora by counting how often which words proceed which histories. Because this training does not require translated sentence pairs, the available corpora are usually much larger, often by orders of magnitude. If the data is sufficiently good, this increase in training data can improve the model's accuracy.

When using this approach, it is important to notice that during the translation, the risk of encountering unknown n-grams is large. For $n = 3$ and a vocabulary of size 20,000 , there are $20{,}000^3 = 8{\cdot}10^{12}$ possible n-grams. Even though most of these n-grams will be meaningless, it is very likely that even a large training corpus does not contain all n-grams that can possibly be used. These unseen n-grams would have a probability of zero, causing the whole chain of multiplications in Equation 2.6 to become zero as well. Therefore, every translation that contains even a single unseen n-gram would be considered completely impossible by the algorithm. To avoid this issue, one usually applies modified Kneser-Ney smoothing as introduced by Kneser and Ney [1995], Chen and Goodman [1996]. This smoothing technique assigns part of the probability mass to unseen n-grams.

Brown et al. [1993] propose to use additional hidden variables a_1^J that provide alignment information from the target to the source sentence. Using those alignments, they further split the inverted translation model $\Pr(f_1^J|e_1^I)$.

$$\Pr(f_1^J|e_1^I) = \sum_{a_1^J \in A} \Pr(f_1^J, a_1^J|e_1^I) \tag{2.7}$$

$$= \sum_{a_1^J \in A} \big(\Pr(J|e_1^I) \cdot \Pr(f_1^J, a_1^J|e_1^I, J)\big) \tag{2.8}$$

$$= \sum_{a_1^J \in A} \big(\Pr(J|e_1^I) \cdot \Pr(a_1^J|e_1^I, J) \cdot \Pr(f_1^J|a_1^J, e_1^I, J)\big) \tag{2.9}$$

In these formulas, A is the set of all possible alignments. The new models are called the *sentence length model*, *alignment model* and *lexicon model* respectively. This approach to SMT is visualized in Figure 2.1. Often, the alignment model is used to identify sequences of words that can be translated as a phrase. This process is called *phrase-based translation*.

Usually, the maximization is performed in log-space. Och and Ney [2002] describe how to change the formula to

$$\hat{e}_1^{\hat{I}} = \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \Pr(e_1^I | f_1^J) \right\} \tag{2.10}$$

$$= \underset{I, e_1^I \in E}{\operatorname{argmax}} \left\{ \sum_{m=1}^{M} \lambda_m h_m(e_1^I, f_1^J) \right\} \tag{2.11}$$

where λ_m is a scaling factor and $h_m(e_1^I, f_1^J)$ is a feature function. This formula contains Equation 2.4 as a special case for $M = 2$, $\lambda_1 = \lambda_2 = 1$ and

$$h_1(e_1^I, f_1^J) = \log \Pr(e_1^I) \tag{2.12}$$

$$h_2(e_1^I, f_1^J) = \log \Pr(f_1^J | e_1^I) \tag{2.13}$$

This way of solving the maximization allows to easily add additional feature functions.

2.2 Beam Search

Once all probability distributions or feature functions are learned, they can be used to find translations for unseen source sentences. As a naive way, one could compute all probabilities for all possible target sequences up to some chosen length in order to exactly determine how likely all possible translations are. However, this is computationally infeasible because the number of necessary computations increases exponentially. Let us assume a vocabulary with size 20,000 and a sequence of length 30. For this setup, one would have to perform $20{,}000^{30} \approx 10^{129}$ computations. Therefore, the number of needed computations has to be decreased dramatically.

To do so, only a small portion of all possible translations is explored. This is achieved by using the beam search algorithm. It keeps track of a *beam* of hypotheses. After extending all partial hypotheses, it restricts the beam to the n most likely ones. This process is repeated until n complete hypotheses are found. A pseudo-code can be found in Algorithm 1.

The special case of $n = 1$ is the greedy translation method, where at every time step the translation is extended with the single most likely target word. Even

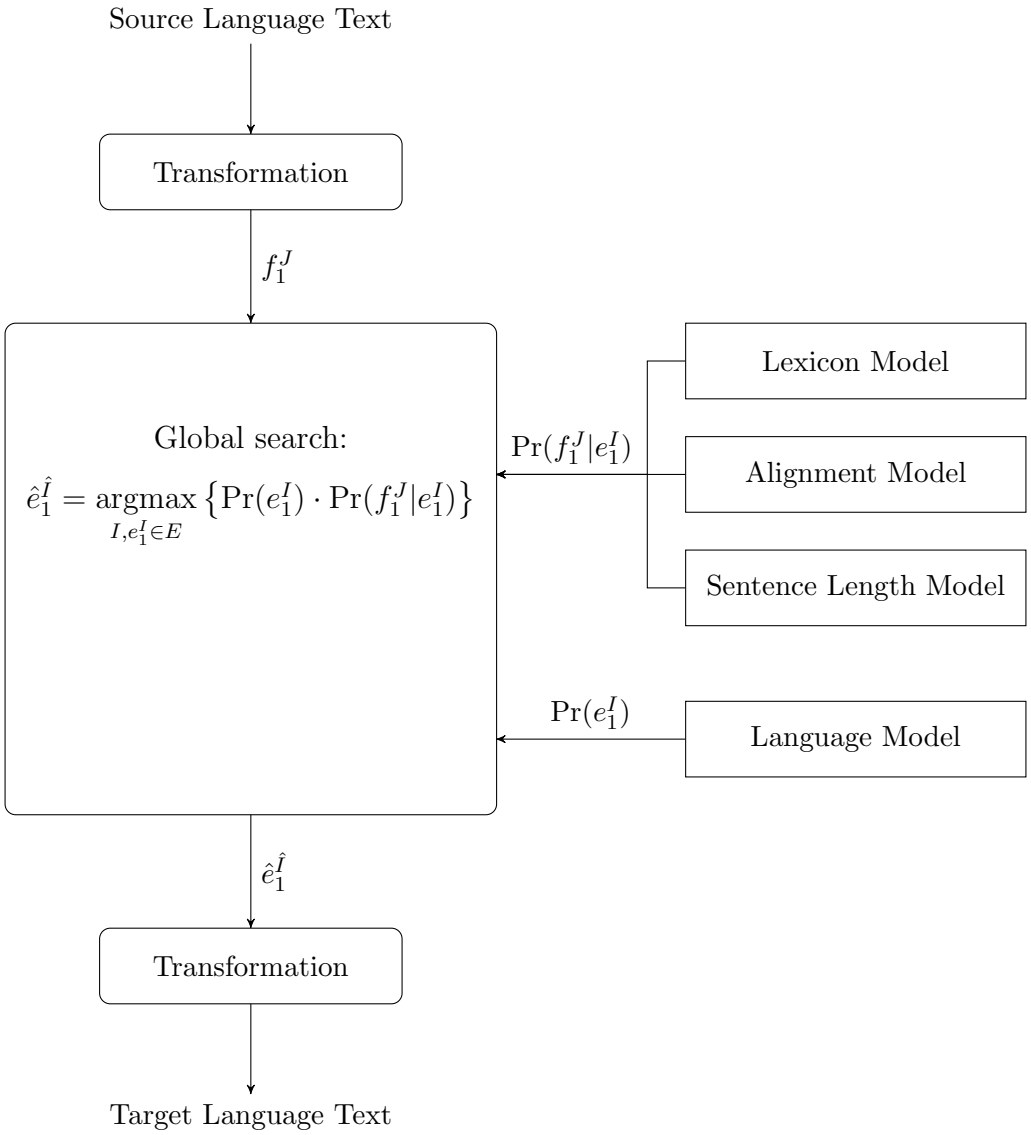


Figure 2.1: The architecture of an SMT system. Using a lexicon model, an alignment model and a sentence length model, $\Pr(f_1^J|e_1^I)$ can be estimated. In combination with the estimation of $\Pr(e_1^I)$ by the LM, the most likely target sentence $\hat{e}_1^{\hat{I}}$ can be determined. Image modified based on material of the chair i6 at RWTH Aachen.

Algorithm 1 A pseudo-code of the beam search algorithm

Require: n: beam size
Require: V: target vocabulary
Require: f_1^J: source sentence
Require: $p(\cdot|\cdot)$: trained model for $\Pr(\hat{e}_1^{\hat{I}}|f_1^J)$

```
1: procedure BEAM SEARCH(n, V, f_1^J, p)
2:     t ← 0
3:     B_t ← [(1, <BOS>)] // the current beam
4:     while |B_t.whereHypothesisComplete| < n do
5:         t ← t + 1
6:         B_t ← []
7:         for (oldProb, partialHyp) ∈ B_{t-1}.whereHypothesisIncomplete do
8:             for v ∈ V do
9:                 extendedHyp ← [partialHyp; v]
10:                newProbability ← p(extendedHyp|f_1^J)
11:                B_t.append((newProbability, extendedHyp))
12:        B_t ← B_t.orderDescendingByProbability()
13:        B_t ← B_t[0 : n] // limit beam to n most likely hypotheses
14:    return B_t
```

though this would be a very efficient search method, it does not allow to recover from mistakingly chosen words, leading to suboptimal translations. By increasing the beam size n, one can improve the overall probability of the final hypothesis. Choosing a target word w_i at time step i that does not have the highest probability might allow to choose a target word w_{i+1} at time step $i+1$ with much higher probability than otherwise possible.

2.3 Automatic Evaluation Metrics

The evaluation of translations by different MT systems is a difficult task. Given a set of hypotheses for the same source sentence translation, even human experts often find it difficult to order them by quality. Additionally, human evaluation is both expensive and time consuming when it is to be used for a meaningfully large number of hypotheses.

Because the ability to quickly iterate and improve the systems is important, automatic measurements had to be invented. In this work, we will evaluate the performance of our systems using BLEU, TER and perplexity (PPL). Those are the measures most widely used in MT research and do not require human interaction. All three of them evaluate the hypotheses with respect to one or more provided

references. In the following sections, BLEU and TER will be explained. PPL will be introduced in Section 3.4.1.

2.3.1 BLEU

The bilingual evaluation understudy (BLEU), proposed by Papineni et al. [2002], is an automatic measure of the translation quality. It is based on the assumption that the number of n-grams that occur both in the reference and the hypothesized translation is an indicator for the translation quality. The score is computed as the geometric mean of n-gram precisions and a *brevity penalty* that avoids translations that are too short. Even though the original formulas allow for multiple reference translations, we only use a single reference e_1^I in this work.

Given a hypothesis $\hat{e}_1^{\hat{I}}$, the BLEU score can be computed as follows: First, one determines the modified n-gram precisions. For every n-gram w_1^n in the hypothesis, one takes the minimum of the occurrences of w_1^n in e_1^I and $\hat{e}_1^{\hat{I}}$. This clipped count is then divided by the number of n-grams that occur in $\hat{e}_1^{\hat{I}}$, leading to the formula

$$\mathrm{ModPrec}_n(e_1^I, \hat{e}_1^{\hat{I}}) = \frac{\sum_{w_1^n} \min\left\{c(w_1^n, e_1^I), c(w_1^n, \hat{e}_1^{\hat{I}})\right\}}{\sum_{w_1^n} c(w_1^n, \hat{e}_1^{\hat{I}})} \tag{2.14}$$

where $c(a, b)$ is a count function providing the number of occurrences of a in b. The brevity penalty $BP(I, \hat{I})$ is defined as

$$BP(I, \hat{I}) = \begin{cases} 1\,, & \text{if } \hat{I} \geq I \\ \exp(1 - \frac{I}{\hat{I}})\,, & \text{otherwise} \end{cases} \tag{2.15}$$

Finally, the BLEU score is computed as follows:

$$BLEU(e_1^I, \hat{e}_1^{\hat{I}}) = BP(I, \hat{I}) \cdot \exp\left(\sum_{n=1}^{N} w_n \log \mathrm{ModPrec}_n(e_1^I, \hat{e}_1^{\hat{I}})\right) \tag{2.16}$$

$$= BP(I, \hat{I}) \cdot \prod_{n=1}^{N} \mathrm{ModPrec}_n(e_1^I, \hat{e}_1^{\hat{I}})^{w_n} \tag{2.17}$$

where N is the maximum n-gram length and w_n weights the influence of different n-gram lengths. Usually, $N = 4$ and $w_n = \frac{1}{N}$. Instead of a sentence-wise evaluation, the BLEU score is commonly computed over the whole test corpus.

Although there is some criticism by Callison-Burch et al. [2006], most studies, such as [Coughlin, 2003], come to the conclusion that BLEU correlates highly with human evaluations.

2.3.2 Translation Edit Rate (TER)

The TER score, proposed by Snover et al. [2006], is an error metric based on the Levenshtein distance as described in [Levenshtein, 1966]. Unlike BLEU, it does not depend on counting n-grams. Instead, TER is looking at the minimal number of modifications that are needed to transform the hypothesis into one of the references.

The TER score can be defined as

$$TER = \frac{\text{minimal number of edits}}{\text{average number of words in the references}} \tag{2.18}$$

Edits can be insertions, deletions, substitutions and shifts. The first three are applied to individual words, a shift is the movement of a continuous sequence of words within the hypothesis. All kinds of edits are assigned an equal cost.

Because determining the exact TER score would require exponential computing power, a greedy approach can be applied. Snover et al. [2006] not only define this approach but also introduce a human-targeted TER that measures the translation quality more realistically by using human assistance. However, due to the high costs associated with any human interaction in the evaluation, this score is not commonly used. Similar to BLEU, TER scores are computed based on the whole test corpus.

Chapter 3

Artificial Neural Network (ANN)

ANNs try to imitate the properties of human or animal brains. While certainly different parts of the brain are responsible for different tasks, these responsibilities are loosely defined. Furthermore, all parts consist of the same substructures, namely neurons. ANNs equally consist of many individual artificial neurons.

3.1 Neuron

A neuron is the most simple individual part of any ANN. It has been proposed by Rosenblatt [1957] who has used them to create a so-called *simple perceptron*, a network with one layer. Similar to actual neurons in brains, they receive input from each other. Once the incoming signals are strong enough, the neuron outputs a signal itself, that is then consumed by other neurons. The threshold that defines when a neuron starts to fire can vary between different neurons and is defined by an *activation function*.

In its original version, the output is always either 0 or 1. Such an activation function f is defined by a weight w and a threshold b. If and only if the input x multiplied by the weight surpasses the threshold, the activation function emits 1.

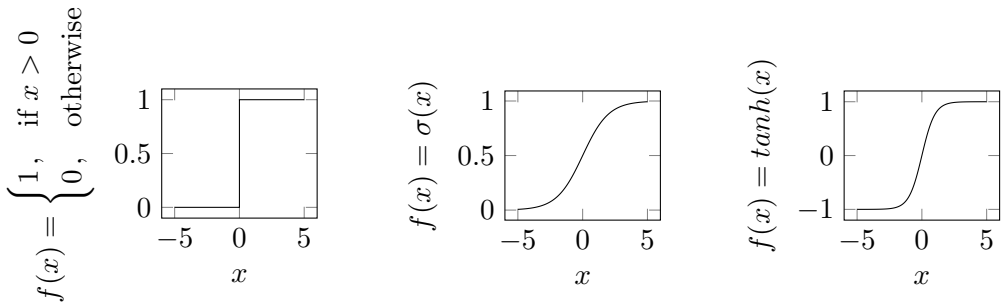
$$f(x) = \begin{cases} 1\,, & \text{if } wx > b \\ 0\,, & \text{otherwise} \end{cases} = \begin{cases} 1\,, & \text{if } wx - b > 0 \\ 0\,, & \text{otherwise} \end{cases} \tag{3.1}$$

As an alternative to this step function, continuous functions can be used. Simple and widely used non-linear functions are:

$$f(x) = \frac{1}{1 + \exp(-wx - b)} = \sigma(wx + b) \tag{3.2}$$

$$f(x) = \tanh(wx + b) \tag{3.3}$$

Figure 3.1 visualizes that these functions closely resemble the step function from Equation 3.1.



(a) A step function with $w = 1$ and $b = 0$

(b) The sigmoid function

(c) The tanh function

Figure 3.1: Activation functions

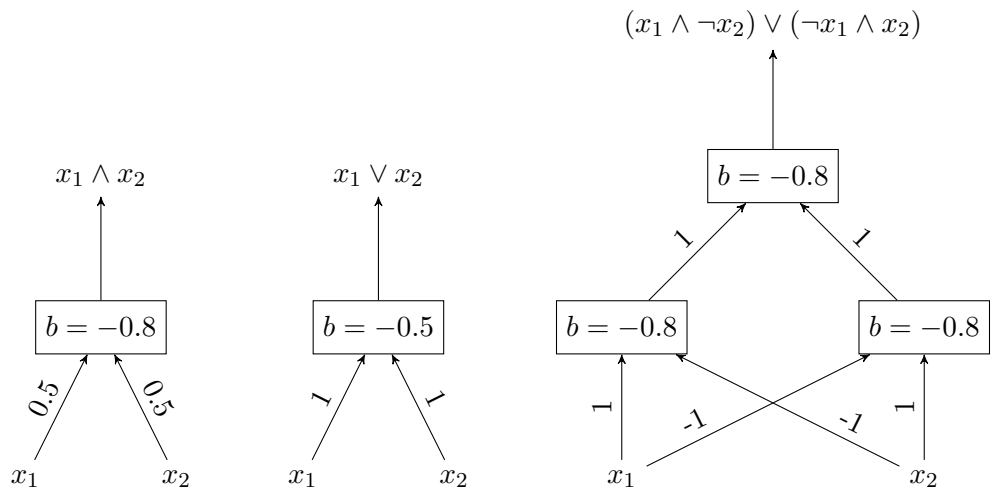
3.2 Feedforward Neural Network (FFNN)

A neuron accepts an input and produces an output that can in turn be processed by another neuron. Because the information constantly flows forward, these networks are categorized as FFNNs. An FFNN with only one layer of neurons can only apply a linear function on the input. Therefore, a network with no hidden layers can only solve problems that are linearly separable. While simple logical operations like AND and OR can be implemented, XOR requires a deeper network. In addition to the input and output layer, a hidden layer has to be added. Because now the output layer can apply a linear function to the non-linear output of the hidden layer, more complex functions are representable. Exemplary implementations of the logical AND, OR and XOR operations are shown in Figure 3.2.

Whether one single hidden layer is enough to allow representing all functions was unclear until 1989, when Hornik et al. [1989], Cybenko [1989] proved that this is indeed possible. This insight is known as the *universal approximation theorem*.

Theorem 1 (Universal Approximation Theorem)
Any feedforward network with one hidden layer using an arbitrary squashing function is capable of approximating any Borel measurable function from one finite dimensional space to another to any desired degree of accuracy, provided sufficiently many hidden units are available.

As this theorem states, any function can be approximated, even though this might require a large number of hidden units. For instance, Coates et al. [2011] train an image recognition network with only one single hidden layer that is as good as the state of the art. In practice, networks with more layers are used, because such a deeper network requires less neurons in total. Once a certain number of layers is



(a) AND orperation (b) OR operation (c) XOR operation

Figure 3.2: Single- and multilayer network implementations of the logical AND, OR and XOR operations. The activation function is a step function. The weights are written on the incoming arrows, the biases are written inside of the neurons.

reached, the network is often called a *deep* neural network (DNN). However, there is no specific threshold for this.

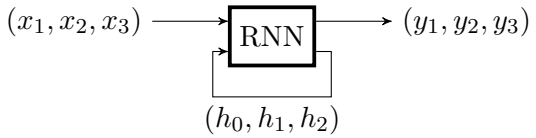
Because layers usually consist of more than one single neuron, defining their formulas as in Equations 3.1 to 3.3 would become prohibitively complicated as every neuron would require its own formula. Instead, multiple neuron values $x_1, \ldots, x_n$ as well as the biases $b_1, \ldots, b_n$ that are in the same layer are summarized in one single vector $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{b} \in \mathbb{R}^n$, respectively. If all neurons use a sigmoid function to determine their activation value, they can be described as

$$f(\boldsymbol{x}) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{3.4}$$

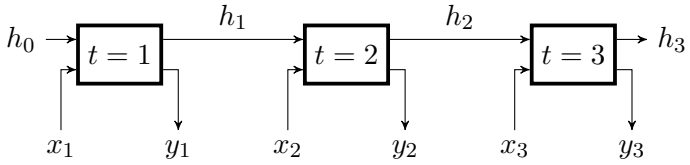
where $\boldsymbol{W} \in \mathbb{R}^{m,n}$ is a weight matrix, defining the weights of all m incoming connections for all n neurons in this layer. $\boldsymbol{b}$ is the bias vector and $\boldsymbol{x}$ is the input vector. For better readability, the bias is often not written explicitly.

3.3 Recurrent Neural Network (RNN)

FFNNs have two inherent problems: They can not be applied to inputs of arbitrary length and they can not remember previous inputs. However, sentences can be long,



(a) Original RNN



(b) Unrolled version of the RNN for 3 time steps

Figure 3.3: Unrolling of an RNN. In the unrolled version, all layers have the same activation function and weight matrix.

and splitting them into separately processed inputs removes part of their context. To avoid this issue, RNNs can be used. They differ from FFNNs such that their output is fed back to their input. Therefore, the network is able to evaluate the current input in the context of previously processed data. Because the activation functions and weight matrices of RNNs are shared across all time steps, they can even generalize to unseen sequence lengths.

Instead of feeding the whole output back to the input, an additional internal memory $\boldsymbol{h}_t$ can be added to the network. The resulting model can be described as

$$\boldsymbol{h}_t = a(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_i \boldsymbol{x}_t) \tag{3.5}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_y \boldsymbol{h}_t \tag{3.6}$$

for a nonlinear function a and weight matrices $\boldsymbol{W}_h$, $\boldsymbol{W}_i$ and $\boldsymbol{W}_y$. $\boldsymbol{x}_1^T$ is the input sequence. Every RNN can be converted to an FFNN for a given length of recurrence. For this, the layers are duplicated as often as needed. This implies that RNNs are a special case of DNNs. The process of mapping the RNN to an FFNN is called *unrolling* and is shown in Figure 3.3. In theory, such an RNN is able to store information in $\boldsymbol{h}_t$ indeterminately until it is needed at a later time step. However, optimizing the weight matrix in a stable way that allows long-term dependencies to have a sufficiently high influence on the network's behavior can be quite difficult. This is caused by the problem of exploding or vanishing gradients during backpropagation through time (BPTT) (see Section 3.4.2). As the error is propagated backwards through multiple time steps, it is repeatedly multiplied by the weights. This causes the error to

1. increase exponentially if the weights are larger than 1

2. decrease exponentially if the weights are smaller than 1

For exploding gradients, the training will be very unstable. Vanishing gradients on the other hand provide only little insight into the correct direction for the parameter updates (see Section 3.4.2). Therefore, training them to minimize the loss value is difficult. Hochreiter et al. [2001] show that an RNN as defined in Equations 3.5 and 3.6 necessarily either suffers from vanishing gradients or is unable to learn long-term dependencies in the data. For a more detailed analysis of RNNs see [Hochreiter, 1991]. In the following sections, we address some approaches that overcome the challenges mentioned here.

3.3.1 Long Short-Term Memory (LSTM)

As stated above, RNNs have difficulties learning long-term dependencies. Hochreiter and Schmidhuber [1997] have proposed a gate-based RNN, called LSTM, to circumvent this issue. Because multiple variations of LSTMs exist and listing them all would go beyond the scope of this work, the following description is based on the setup described in [Graves, 2013] that is most widely used and includes the extension by Gers et al. [2000]. However, in the following equations, we omit the biases for simplicity.

The LSTM has an internal state $\boldsymbol{c}_t$. At every time step t, only minimal adjustments are made to this state, ensuring that the gradient does not vanish. To be concrete, the state $\boldsymbol{c}_t$ is computed as

$$\boldsymbol{c}_t = \begin{cases} \mathbf{0}\,, & \text{if } t = 0 \\ \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \tilde{\boldsymbol{c}}_t\,, & \text{otherwise} \end{cases} \tag{3.7}$$

Where $\tilde{\boldsymbol{c}}_t$ is known as the *input candidate*, containing new information. The *forget gate* $\boldsymbol{f}_t$ controls which parts of the previous state should be forgotten and the *input gate* $\boldsymbol{i}_t$ indicates which parts of the current input candidate should be added to the state.

If $\boldsymbol{i}_t = \mathbf{0}$ and $\boldsymbol{f}_t = \mathbf{1}$, then $\boldsymbol{c}_t = \boldsymbol{c}_{t-1}$ and the error flow is constant. This way, an LSTM can store information for arbitrarily long time intervals without suffering from exploding or vanishing gradients. As described in [Hochreiter and Schmidhuber, 1997], $\boldsymbol{c}_t$ is also called the *constant error carousel* (CEC) of the network. The output $\boldsymbol{h}_t$ is defined as

$$\boldsymbol{h}_t = \begin{cases} \mathbf{0}\,, & \text{if } t = 0 \\ \tanh\left(\boldsymbol{c}_t\right) \circ \boldsymbol{o}_t\,, & \text{otherwise} \end{cases} \tag{3.8}$$

Here, $\boldsymbol{o}_t$ is an *output gate*. The gates are computed using sigmoid functions as follows:

$$\boldsymbol{o}_t = \sigma\Big(\boldsymbol{W}_o\boldsymbol{x}_t + \boldsymbol{U}_o\boldsymbol{h}_{t-1} + \boldsymbol{V}_o\boldsymbol{c}_t\Big) \tag{3.9}$$

$$\boldsymbol{i}_t = \sigma\Big(\boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{U}_i\boldsymbol{h}_{t-1} + \boldsymbol{V}_i\boldsymbol{c}_{t-1}\Big) \tag{3.10}$$

$$\boldsymbol{f}_t = \sigma\Big(\boldsymbol{W}_f\boldsymbol{x}_t + \boldsymbol{U}_f\boldsymbol{h}_{t-1} + \boldsymbol{V}_f\boldsymbol{c}_{t-1}\Big) \tag{3.11}$$

The input candidate is similarly defined:

$$\tilde{\boldsymbol{c}}_t = \tanh\Big(\boldsymbol{W}_c\boldsymbol{x}_t + \boldsymbol{U}_c\boldsymbol{h}_{t-1}\Big) \tag{3.12}$$

In Equations 3.9 to 3.11 the additions of the cell state $\boldsymbol{c}_t$ and $\boldsymbol{c}_{t-1}$, respectively, are called *peepholes*. They are not part of the original definition of LSTMs but were later added by Gers and Schmidhuber [2000] to allow the LSTM to access its current state during the gate computation. The weight matrices $\boldsymbol{V}_o$, $\boldsymbol{V}_i$ and $\boldsymbol{V}_f$ are diagonal. A graphical representation of an LSTM cell can be seen in Figure 3.4a. To increase the readability, we make the following definitions:

1. $\boldsymbol{M}_g = [\boldsymbol{W}_g, \boldsymbol{U}_g, \boldsymbol{V}_g] \, \forall g \in \{o, i, f\}$
2. $\boldsymbol{M}_c = [\boldsymbol{W}_c, \boldsymbol{U}_c]$
3. $\boldsymbol{c}_0 = \boldsymbol{h}_0 = 0$, to avoid the case differentiations

The final equations for the forward pass of the LSTM in the order of their computation are therefore:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T) \tag{3.13}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T) \tag{3.14}$$

$$\tilde{\boldsymbol{c}}_t = \tanh\left(\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T\right) \tag{3.15}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \tilde{\boldsymbol{c}}_t \tag{3.16}$$

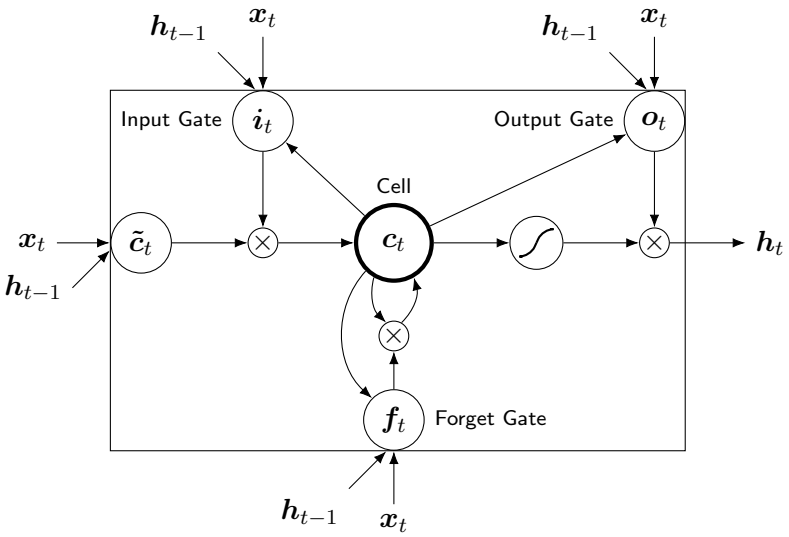
$$\boldsymbol{o}_t = \sigma(\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T) \tag{3.17}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh\left(\boldsymbol{c}_t\right) \tag{3.18}$$

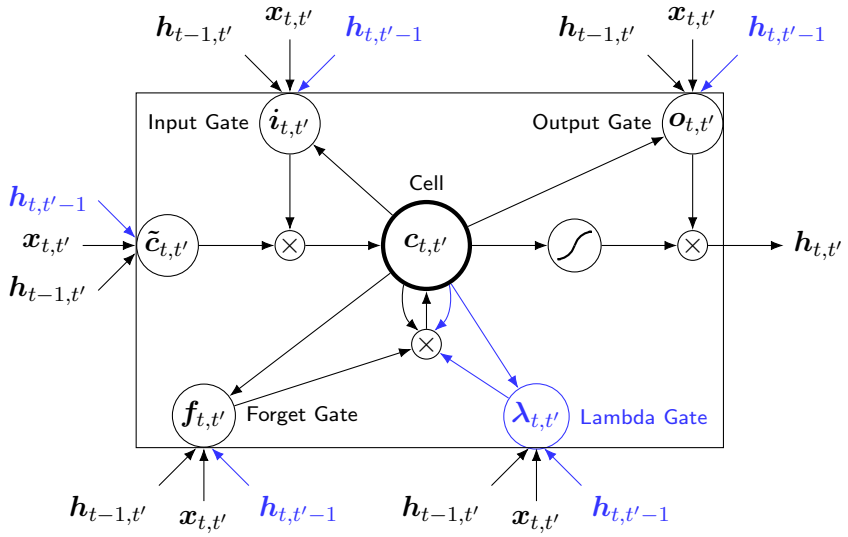
3.3.2 Multi-Dimensional LSTM (MDLSTM)

The LSTM introduced in Section 3.3.1 processes a stream of vectors, one at a time. This is helpful in use cases where the input can readily be represented as a one-dimensional (1D) stream of data, such as the words within a sentence.

However, some data naturally has more than one dimension. Images, for example, cannot easily be transformed to meaningful 1D data point sequences. Fortunately,

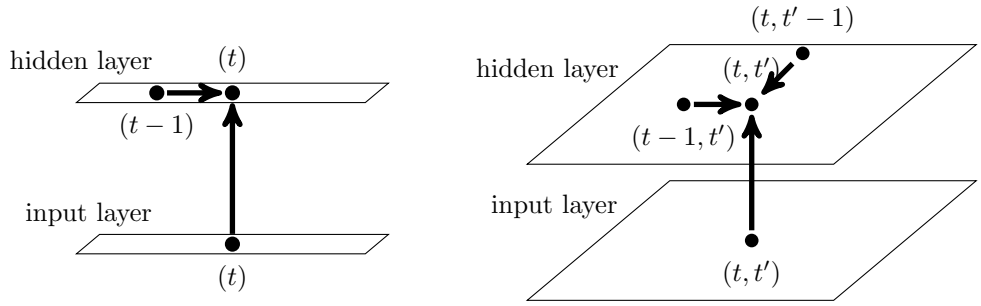


(a) An LSTM cell



(b) A two-dimensional LSTM (2DLSTM) cell

Figure 3.4: An overview of an LSTM and a 2DLSTM cell. The additional elements and connections in the 2DLSTM cell are highlighted in blue. Figure adapted based on [Graves, 2013].



(a) Dependencies in a 1D RNN (b) Dependencies in a two-dimensional (2D) RNN

Figure 3.5: Dependencies in a 1D and 2D RNN. For two dimensions, the state depends on two previous time steps: $(t-1,t')$ and $(t,t'-1)$. Figure adapted from [Graves et al., 2007].

an LSTM can be extended to process multi-dimensional inputs. Graves et al. [2007] propose the concept of a multi-dimensional RNN in general and an MDLSTM in particular.

The time axes of a 2DLSTM extend in two directions. Therefore, the current time step has to be specified by two indices. In this work, t and t' represent the horizontal and vertical axes respectively. At each time step (t,t'), a 2DLSTM does not only depend upon its previous state in $(t-1,t')$, but on the state at $(t,t'-1)$ as well, as depicted in Figure 3.5. A visualization of a 2DLSTM cell can be found in Figure 3.4b.

For 2DLSTM, this adaptation results in the following formulas:

$$\boldsymbol{h}_{t,t'} = \begin{cases} \boldsymbol{0}\,, & \text{if } t = 0 \vee t' = 0 \\ \tanh\left(\boldsymbol{o}_{t,t'} \circ \boldsymbol{c}_{t,t'}\right)\,, & \text{otherwise} \end{cases} \tag{3.19}$$

$$\boldsymbol{o}_{t,t'} = \sigma\Big(\boldsymbol{W}_o\boldsymbol{x}_{t,t'} + \boldsymbol{U}_o\boldsymbol{h}_{t-1,t'} + \boldsymbol{U}'_o\boldsymbol{h}_{t,t'-1} + \boldsymbol{V}_o\boldsymbol{c}_{t,t'}\Big) \tag{3.20}$$

$$\boldsymbol{i}_{t,t'} = \sigma\Big(\boldsymbol{W}_i\boldsymbol{x}_{t,t'} + \boldsymbol{U}_i\boldsymbol{h}_{t-1,t'} + \boldsymbol{U}'_i\boldsymbol{h}_{t,t'-1} + \boldsymbol{V}_i\boldsymbol{c}_{t-1,t'} + \boldsymbol{V}'_i\boldsymbol{c}_{t,t'-1}\Big) \tag{3.21}$$

$$\boldsymbol{f}_{t,t'} = \sigma\Big(\boldsymbol{W}_f\boldsymbol{x}_{t,t'} + \boldsymbol{U}_f\boldsymbol{h}_{t-1,t'} + \boldsymbol{U}'_f\boldsymbol{h}_{t,t'-1} + \boldsymbol{V}_f\boldsymbol{c}_{t-1,t'} + \boldsymbol{V}'_f\boldsymbol{c}_{t,t'-1}\Big) \tag{3.22}$$

$$\boldsymbol{\lambda}_{t,t'} = \sigma\Big(\boldsymbol{W}_\lambda\boldsymbol{x}_{t,t'} + \boldsymbol{U}_\lambda\boldsymbol{h}_{t-1,t'} + \boldsymbol{U}'_\lambda\boldsymbol{h}_{t,t'-1} + \boldsymbol{V}_\lambda\boldsymbol{c}_{t-1,t'} + \boldsymbol{V}'_\lambda\boldsymbol{c}_{t,t'-1}\Big) \tag{3.23}$$

$$\tilde{\boldsymbol{c}}_{t,t'} = \tanh\Big(\boldsymbol{W}_c\boldsymbol{x}_{t,t'} + \boldsymbol{U}_c\boldsymbol{h}_{t-1,t'} + \boldsymbol{U}'_c\boldsymbol{h}_{t,t'-1}\Big) \tag{3.24}$$

$$\boldsymbol{c}_{t,t'} = \begin{cases} \boldsymbol{0}\,, & \text{if } t = 0 \vee t' = 0 \\ \boldsymbol{f}_{t,t'} \circ \left(\boldsymbol{\lambda}_{t,t'} \circ \boldsymbol{c}_{t-1,t'} + (\boldsymbol{1} - \boldsymbol{\lambda}_{t,t'}) \circ \boldsymbol{c}_{t,t'-1}\right) + \boldsymbol{i}_{t,t'} \circ \tilde{\boldsymbol{c}}_{t,t'}\,, & \text{otherwise} \end{cases} \tag{3.25}$$

An additional lambda gate $\boldsymbol{\lambda}_{t,t'}$ (see Equation 3.23) is introduced. As written in Equation 3.25, it controls whether the 2DLSTM focuses on the previous cell state $\boldsymbol{c}_{t-1,t'}$ or on $\boldsymbol{c}_{t,t'-1}$.

After the whole 2D input has been processed, one usually has to reduce the computed cell states $\boldsymbol{c}_{t,t'} \in \mathbb{R}^{T \times T'}$ to a sequence of vectors or even a single vector, in order to process the output at a higher layer. This can be done by

1. summing over one of the axes, as done by Graves and Schmidhuber [2008]
2. taking the last row or column ($\boldsymbol{c}_{1,T'}, \ldots, \boldsymbol{c}_{T,T'}$ or $\boldsymbol{c}_{T,1}, \ldots, \boldsymbol{c}_{T,T'}$)

In the former case, to get a single vector, one can use an additional LSTM. In the latter, one can directly use $\boldsymbol{c}_{T,T'}$. In theory, this state should contain the information of the whole input, because it has been completely processed at this point in time.

For notational simplicity, one can again make the following definitions:

1. $\boldsymbol{M}_g = [\boldsymbol{W}_g, \boldsymbol{U}_g, \boldsymbol{U}'_g, \boldsymbol{V}_g, \boldsymbol{V}'_g] \, \forall g \in \{i, f, \lambda\}$
2. $\boldsymbol{M}_o = [\boldsymbol{W}_o, \boldsymbol{U}_o, \boldsymbol{U}'_o, \boldsymbol{V}_o]$
3. $\boldsymbol{M}_c = [\boldsymbol{W}_c, \boldsymbol{U}_c, \boldsymbol{U}'_c]$
4. $\boldsymbol{c}_{0,s} = \boldsymbol{c}_{s,0} = \boldsymbol{h}_{0,s} = \boldsymbol{h}_{s,0} = 0 \, \forall s \in \mathbb{N}$

The equations can therefore be written in the more compact form:

$$\boldsymbol{f}_{t,t'} = \sigma(\boldsymbol{M}_f[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t-1,t'}, \boldsymbol{c}^T_{t,t'-1}]^T) \tag{3.26}$$

$$\boldsymbol{i}_{t,t'} = \sigma(\boldsymbol{M}_i[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t-1,t'}, \boldsymbol{c}^T_{t,t'-1}]^T) \tag{3.27}$$

$$\boldsymbol{\lambda}_{t,t'} = \sigma(\boldsymbol{M}_\lambda[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t-1,t'}, \boldsymbol{c}^T_{t,t'-1}]^T) \tag{3.28}$$

$$\tilde{\boldsymbol{c}}_{t,t'} = \tanh(\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T) \tag{3.29}$$

$$\boldsymbol{c}_{t,t'} = \boldsymbol{f}_{t,t'} \circ (\boldsymbol{\lambda}_{t,t'} \circ \boldsymbol{c}_{t-1,t'} + (\mathbf{1} - \boldsymbol{\lambda}_{t,t'}) \circ \boldsymbol{c}_{t,t'-1}) + \boldsymbol{i}_{t,t'} \circ \tilde{\boldsymbol{c}}_{t,t'} \tag{3.30}$$

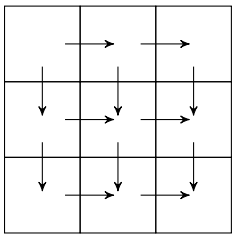
$$\boldsymbol{o}_{t,t'} = \sigma(\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T) \tag{3.31}$$

$$\boldsymbol{h}_{t,t'} = \boldsymbol{o}_{t,t'} \circ \tanh(\boldsymbol{c}_{t,t'}) \tag{3.32}$$

The 2DLSTM cells used in our experiments do not have peephole connections.

Implementation

To develop our models, we use our in-house implementation for neural machine translation (NMT) that is based on the open source library Theano [Theano Development Team, 2016] and the framework Blocks [van Merriënboer et al., 2015].



(a) Dependencies

1	2	3
4	5	6
7	8	9

(b) Ordered Processing

1	2	3
2	3	4
3	4	5

(c) Parallel Processing

Figure 3.6: Parallelization of a 2DLSTM. (a) depicts the dependencies of each cell in the 2DLSTM, the arrows indicate the necessary flow of information. (b) visualizes a possible order to compute the cell values. (c) is a parallelized computation order. As the cells in the minor diagonals do not depend upon each other, they can be computed in parallel. Figure adapted from [Voigtlaender et al., 2016].

Theano optimizes the execution speed of training and decoding by using a computational graph. This graph is created once at the setup of the experiment and determines the specific order of mathematical operations. Theano then optimizes this graph with respect to speed and computational stability. It will eg. completely remove long computations, that are multiplied by 0. This characteristic allows high flexibility during development while still ensuring fast execution.

A further speedup can be gained by utilizing the graphics processing unit (GPU) as it is built to support very fast and parallel matrix operations. Theano takes care of this for most operations automatically and replaces them with their respective GPU implementation. However, Theano does not support MDLSTM, so their computation would mostly rely on the central processing unit (CPU).

The RWTH extensible training framework for universal recurrent neural networks (RETURNN) [Doetsch et al., 2016] solves this problem. It offers an implementation of multidirectional 2DLSTM that does not only heavily utilize the GPU but also parallelizes the computation of different 2DLSTM-cells. If the 2DLSTM is computed from the topmost left corner to the rightmost bottom corner, all cells in the minor diagonals are independent from each other and can therefore be computed simultaneously. This optimization is described in [Voigtlaender et al., 2016] and visualized in Figure 3.6. It reduces the computational complexity of an $n \times m$ 2DLSTM from $\mathcal{O}(nm)$ to $\mathcal{O}(n + m)$.

3.3.3 Gated Recurrent Unit (GRU)

While LSTM solves the problem of long-term dependencies, it is relatively complex. Cho et al. [2014] have proposed the GRU that circumvents the gradient problem just like an LSTM but has a simpler structure. They combine the output $\boldsymbol{h}_t$ and cell state $\boldsymbol{c}_t$ into a single vector $\boldsymbol{h}'_t$:

$$\boldsymbol{h}'_t = \begin{cases} \boldsymbol{0}\,, & \text{if } t = 0 \\ \left(\boldsymbol{1} - \boldsymbol{z}_t\right) \circ \boldsymbol{h}'_{t-1} + \boldsymbol{z}_t \circ \tilde{\boldsymbol{h}'}_t\,, & \text{otherwise} \end{cases} \tag{3.33}$$

where

$$\tilde{\boldsymbol{h}'}_t = \tanh\Big(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h \Big[\boldsymbol{r}_t \circ \boldsymbol{h}'_{t-1}\Big]\Big) \tag{3.34}$$

$$\boldsymbol{r}_t = \sigma\Big(\boldsymbol{W}_r \boldsymbol{x}_t + \boldsymbol{U}_r \boldsymbol{h}'_{t-1}\Big) \tag{3.35}$$

$$\boldsymbol{z}_t = \sigma\Big(\boldsymbol{W}_z \boldsymbol{x}_t + \boldsymbol{U}_z \boldsymbol{h}_{t-1}\Big) \tag{3.36}$$

$\boldsymbol{r}_t$ is called the *reset gate* of the GRU. $\boldsymbol{z}_t$ is a *update gate* that replaces the input and forget gate of an LSTM cell. $\boldsymbol{W}_h$, $\boldsymbol{W}_r$, $\boldsymbol{W}_z$, $\boldsymbol{U}_h$, $\boldsymbol{U}_r$, $\boldsymbol{U}_z$ are weight matrices. Chung et al. [2014] show that a GRU performs as good as LSTM while needing less free parameters, and that both outperform a simple RNN. $\boldsymbol{h}_t$ will be the name of another variable as well (see Section 4.2). To avoid this conflict in naming conventions, the state $\boldsymbol{h}_t$ of the LSTM and $\boldsymbol{h}'_t$ of the GRU will both be referred to as $\boldsymbol{a}_t$ from now on. The notation $\boldsymbol{a}_t = \mathrm{RNN}(\boldsymbol{x}_t, \boldsymbol{a}_{t-1})$ will be used to refer to the output $\boldsymbol{a}_t$ given some input $\boldsymbol{x}_t$ where the previous RNN state was $\boldsymbol{a}_{t-1}$. $\boldsymbol{x}_t$ may be a concatenation of multiple inputs. For LSTM, the internal memory cell is not explicitly written.

3.4 Training

Because common architectures are deep and have large weight matrices, they have several millions of parameters. Once the structure of a neural network is defined, these are initialized with random values. Afterwards, they are trained to improve the performance of the network.

At each step, multiple examples from the training set are bundled in one batch. The network calculates the predicted output $\hat{\boldsymbol{y}}$ for each example in the batch. For *supervised learning*, the correct labels $\boldsymbol{y}$ are known. They can be used to assess how closely the prediction resembles the optimal classification. To this end, a predefined *cost* or *loss function* $\mathcal{L}$ is used. Then, the gradient of this cost function is calculated with respect to the individual network's parameters. This gradient indicates whether the values of the parameters have to be increased or decreased in

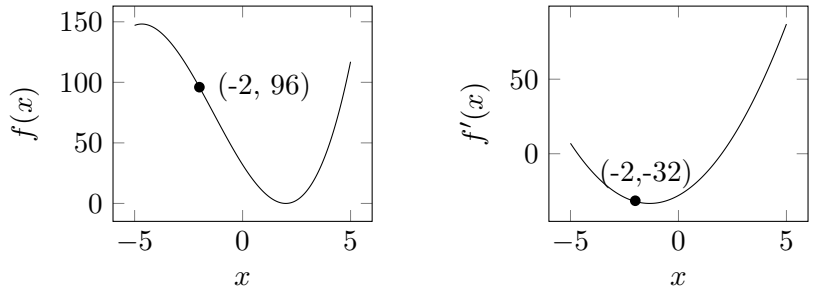


Figure 3.7: The gradient indicates the direction of the update. Because $f'(-2) < 0$, x has to be increased to get closer to the local optimum.

order to reach a lower resulting loss function value. For a positive gradient value, the parameter has to be decreased, for a negative, it has to be increased. Figure 3.7 visualizes this principle. Finally, the parameters are updated into the previously determined directions and a new batch is used to repeat this process. The training is stopped after a predefined number of iterations or once the parameter's values have converged. In addition to the training set, one uses a *development set* to assess the performance of the network. It consists of sentences that are not included in the training set. It is important to stop the training once the performance on the development set starts to decrease. Otherwise, one risks that the model starts to *overfit*. It looses its ability to *generalize* beyond the training samples and instead only memorizes the true labels from the training set. This leads to a worse general performance. To prevent the network from memorizing the true labels in the development or test sets, the sentences in these sets are not used during the training.

In the following sections, we describe the most commonly used loss function and cover the gradient computation for FFNNs and RNNs in detail. Additionally, we describe three different algorithms for the weight updates.

3.4.1 Loss Function

To assess how close the hypothesis of the network is to the *ground truth*, a loss function $\mathcal{L}$ has to be applied. The most common choice for $\mathcal{L}$ is the cross-entropy. For a target and source sentence pair (e, f), it is defined as

$$\mathcal{L}(e, f) = -\log(p(e_n|f_n)) \tag{3.37}$$

For machine translation (MT), Ranzato et al. [2015] list two drawbacks of training using the cross-entropy. Firstly, during the training, the network is supplied

with the first symbols in the ground truth to complete partial hypotheses. In the decoding phase however, the previous output is used to complete partial sequences. Therefore, errors can accumulate. The second issue is that the cross-entropy differs from the final evaluation function. Whereas the cross-entropy is computed word-wise, the scoring functions described in Section 2.3 operate on the whole corpus. Shen et al. [2016] propose *minimum risk training* (MRT) to avoid these problems. They introduce an additional term to the loss function that is based on the expectation of the evaluation metric such as Bleu or Ter. However, cross-entropy is still widely used and is applied in our experiments.

Perplexity (PPL)

The Ppl can be used as a measure of how close the probability distribution computed by the neural network (NN) is to the correct prediction. It is based on the cross-entropy and can be computed as

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{n=1}^{N}\log(p(e_n|f_n))\right) \tag{3.38}$$

where N is the total number of symbols in the reference corpus and e_n and f_n are the target and source sentences, respectively.

The Ppl can be seen as an indicator of how sure the network is about its prediction. If most of the probability mass is focused on the correct prediction, the cost and therefore the Ppl will be low. If the network distributes the probability mass over multiple words, the Ppl will increase, even if the correct target word is still predicted as the most likely one.

Because this measure does not require that the test corpus is actually decoded (as an input, only the source and part of the reference are used), it requires less computational time.

If the cross-entropy and therefore the Ppl has a high value, it means that certain words of the target sequence are falsely given too low probabilities. Therefore, by minimizing the cross-entropy and Ppl, the network is improved. This process is equivalent to maximizing the log-likelihood of the translation.

3.4.2 Optimization

In order to achieve a fast optimization of the NN that ultimately finds a good local optimum, several key aspects are of importance. The initial parameters should be chosen in a way that does not hinder the training process. To obtain information about the needed updates, all functions have to be differentiable. And finally, the weight update should ensure a fast and steady training process. The following sections will explain these aspects in more detail.

Initialization

At the beginning of the training, all parameters of the model have to be initialized. For this, one can choose many distributions. Popular choices are

1. *Random isotropic gaussian initialization*: A normal distribution with mean 0 and a low standard deviation (eg. 0.01). This is the most simple approach as it does not require any knowledge of the network

2. *Xavier initialization*: This initialization has been proposed in [Glorot and Bengio, 2010]. A uniform distribution with the boundaries $\pm\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}$ where n_j and n_{j+1} are the numbers of neurons in this layer and the layer above, respectively. This causes the variance of the gradient to be stable. Otherwise it might increase or decrease, hindering the training. It is sometimes referred to as a *glorot distribution* as well.

After the network is initialized, the parameters are repeatedly updated until a local optimum is reached where the output of the network is close to the correct result.

Whatever initialization is chosen, it is important not to set all weights to the same value, like 0. This would cause the gradient and therefore the updates of all weights to be the same. Only by introducing asymmetry can the network assign different functions to different neurons.

Backpropagation

In order to determine the direction of the parameter updates, the gradient has to be computed. In Section 3.1 the formula for a single neuron is described. Clearly, the simple function $f(x) = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ can easily be derived. The derivation of $\sigma(\boldsymbol{x})_k$ for $\boldsymbol{x} \in \mathbb{R}^K$ and $1 \leq k \leq K$ with respect to $\boldsymbol{x}_k$ is $\sigma(\boldsymbol{x})_k \cdot (1 - \sigma(\boldsymbol{x})_k)$, the derivation of $\boldsymbol{W}\boldsymbol{x}$ with respect to $\boldsymbol{W}$ and $\boldsymbol{x}$ can be found in Appendix A.1.1.

Multiple layers of neurons, as described in Section 3.2, only require chaining multiple neuron computations. To compute the gradients with respect to parameters of deeper layers, a method called *back-propagation* is used. Back-propagation repeatedly applies the chain rule of calculus to compute all necessary gradients one after another. It was popularized by [Rumelhart et al., 1986]. However, the concept in general was already known before.

Theorem 2 (Chain Rule)
For a given function $h(x) = f(g(x))$ with scalar x, the derivative can be defined as $h'(x) = f'(g(x)) \cdot g'(x)$.

When computing the gradients with respect to parameters in RNNs, one has to take the recurrence into account. There, one has to start at the last time step and

move backwards in time until time step 1. Since, in this process, the gradients are propagated backwards through time, this version of back-propagation is called *backpropagation through time* (BPTT). The gradients with respect to parameters within the LSTM and 2DLSTM are derived in Appendices A.1.2 and A.1.3.

Current popular frameworks like Theano [Theano Development Team, 2016] are able to perform this computation automatically.

Weight Update

As described above, after determining in which directions the parameters have to be changed, they are increased or decreased by a small amount $\Delta \boldsymbol{w}_t$.

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} + \Delta \boldsymbol{w}_t \tag{3.39}$$

How large this amount is, determines the speed of convergence as well as the ability to reach a good local optimum. If the update is too small, the network needs to be updated extremely often, increasing the training time. The upside is that the network can exactly follow the slope of the cost function, eventually finding a good local minimum. If the update step size is increased, the network makes larger steps into the determined direction. In the best case, this reduces the training time because the local optimum is found much faster. However, if the step size is too large, one risks overshooting an optimum, meaning that the network might start to oscillate around an optimum or miss it completely. If it only causes the network to settle on another local minimum, the impact is negligible. It is more severe if the overshooting causes the network to climb the hills in parameter space, causing it to perform worse. This situation is shown in Figure 3.8.

Different techniques have been proposed to determine by how much each parameter should be updated, in order to maximize training speed while minimizing the risk of overshooting an optimum. This subsection presents three different optimization techniques. More details can be found in [Ruder, 2016].

Stochastic Gradient Descent (SGD) The most simple option to determine $\Delta \boldsymbol{w}_t$ is to choose it based on the current gradient $\frac{\partial E}{\partial \boldsymbol{w}_t}$ where E is the value of the loss function. A large gradient may indicate that a vast enhancement is possible by drastically changing the parameter. A small gradient may imply that the cost will not decrease much by updating this parameter. This reasoning leads to the following formula:

$$\Delta \boldsymbol{w}_t = -\alpha \cdot \frac{\partial E}{\partial \boldsymbol{w}_t} \tag{3.40}$$

where $\alpha > 0$ is the learning rate, a hyperparameter that is defined at the beginning of the training. Algorithm 2 provides a pseudo-code of SGD.

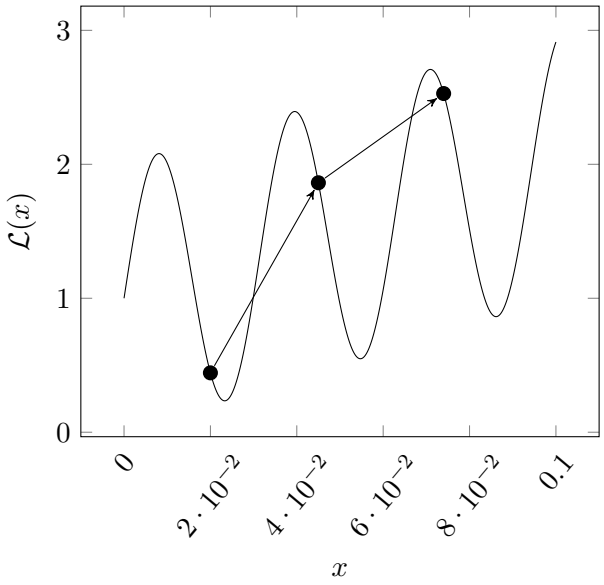


Figure 3.8: Overshooting during parameter optimization. At each update, increasing x appears to be beneficial. By overshooting the local optima, each update causes the network to perform worse.

Algorithm 2 A pseudo-code of SGD.

Require: α: learning rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function
1: **procedure** SGD
2: $t \leftarrow 0$
3: **while** θ_t not converged **do**
4: $t \leftarrow t + 1$
5: $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
6: $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$
7: **return** θ_t

Momentum The default SGD method uses one universal learning rate for all parameters. However, this might not be optimal. If the path to a local optimum leads through a long sharp valley, one parameter (θ_1) has to be changed a lot to slowly reach a better point and even small changes to other parameters (θ_2, θ_3) cause a huge decline in quality. In this situation, the following problem arises [Sutton, 1986].

The absolute value of the gradients of θ_2 and θ_3 will be large, causing equally large updates to these parameters. This carries the danger of overshooting and losing the path to the local optimum. The absolute value of the gradient of θ_1 on the other hand will be small, causing infinitesimal updates and improvements. In order to minimize the risk of overshooting the walls of the valley, the learning rate has to be reduced, slowing down the training even more. The updates will thereby mostly consist of oscillating between the two valley walls with minimal progress towards its end. This problem can be resolved by using momentum as proposed by Rumelhart et al. [1988]. They define the weight update to be determined as

$$\Delta \boldsymbol{w}_t = \beta \cdot \Delta \boldsymbol{w}_{t-1} - \alpha \cdot \frac{\partial E}{\partial \boldsymbol{w}_t} \tag{3.41}$$

for a given learning rate $\alpha > 0$ and decay rate $0 \leq \beta < 1$.

Instead of computing the parameter update solely based on the current gradient, previous gradients are taken into account as well. If multiple updates in the same direction have been applied consecutively, the update for this parameter is increased. However, if the direction of the gradient changes regularly, they cancel each other out and the momentum $\beta \cdot \Delta \boldsymbol{w}_{t-1}$ is small, reducing the update size. At the same time, the effect of previous gradients decreases exponentially with every time step, because they are repeatedly multiplied with β. A pseudo-code of momentum is shown in Algorithm 3.

Adaptive Moment Estimation (Adam) Adam has been proposed by Kingma and Ba [2014] as an adaptive optimization technique. In addition to storing the exponentially decaying gradient of the past iterations like momentum, it also computes an exponentially decaying average of the past squared gradients. They are estimations of the first and second moment of the gradients. Because these averages are initialized with zeros, they are biased to zero during the first iterations. This issue becomes larger when the decay rate of previous gradients is chosen to be small. This is counterbalanced by bias-correcting them, as can be seen in the pseudo-code in Algorithm 4.

The improvement of Adam over momentum is its ability to approximately limit the step size to the learning rate value, avoiding too large updates to the parameters. However, Reddi et al. [2018] recently showed that Adam does not necessarily

Algorithm 3 A pseudo-code of momentum

Require: α: learning rate
Require: $\beta \in [0, 1)$: decay rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function

procedure ADAM
    $w_0 \leftarrow 0$
    $t \leftarrow 0$
    while θ_t not converged **do**
        $t \leftarrow t + 1$
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
        $\Delta w_t \leftarrow \beta \cdot \Delta w_{t-1} - \alpha \cdot g_t$
        $\theta_t \leftarrow \theta_{t-1} + \Delta w_t$
    return θ_t

Algorithm 4 A pseudo-code of Adam. Taken from [Kingma and Ba, 2014]

Require: α: learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$: decay rate
Require: θ_0: parameter set
Require: $f(\theta)$: network function

procedure ADAM
    $m_0 \leftarrow 0$
    $v_0 \leftarrow 0$
    $t \leftarrow 0$
    while θ_t not converged **do**
        $t \leftarrow t + 1$
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
        $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
        $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
        $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
        $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$
    return θ_t

converge to an optimum. As this is a new finding and Adam is currently often used, we applied Adam in our experiments (see Chapter 6).

3.5 Practical Observations

Once a convergence is noted, there are several possibilities to further improve the network. One can, for example:

1. Continue the training with a decreased learning rate, thereby allowing the network to finely adjust its parameter values. If this is done repeatedly, it is called *annealing* (see eg. [Bahar et al., 2017]).

2. Average the parameters of multiple snapshots from a single training run.

For the second option, one has to save multiple snapshots of the model during the training process. This technique was first reported by Utans [1996] and then rediscovered by J.-D. et al. [2016]. It is likely that the n snapshots of a single training run all cluster around the same local optimum. By averaging all parameters of the models, individual small errors can be evened out.

Further improvement is possible by training multiple different models and building an *ensemble*. In an ensemble, the scores of all used models are combined at every time step to generate the final hypothesis [Hashem and Schmeiser, 1995]. This has the benefit of reducing the output's variance and thereby increasing the accuracy. The models do not necessarily need to be trained on the same data and may have a different internal structure. However, they have to use the same output vocabulary.

Chapter 4

Neural Machine Translation (NMT)

The models and techniques explained in Chapter 3 can be applied to the field of machine translation (MT). In the following sections, we will describe commonly used network architectures for NMT.

4.1 Encoder-Decoder Architecture

Many neural network (NN) architectures require to limit the context of sequences. This is in conflict with the nature of languages, because sentences can be unrestrictedly long.

To avoid having to define a maximum context length like in the feedforward neural network (FFNN) language model (LM) in [Bengio et al., 2003], the most common NMT systems are based on the encoder-decoder architecture. It is composed of two long short-term memory (LSTM) cells, as described in [Sutskever et al., 2014, Cho et al., 2014]. Given a source sequence $f_1^J = f_1, \ldots, f_J$ and a target sequence $e_1^I = e_1, \ldots, e_I$, the encoder of the encoder-decoder strategy reads the source sequence and aims to encode it into a set of vectors. The decoder then generates the variable-length target sequence. A similar technique is described in [Kalchbrenner and Blunsom, 2013], even though they use a convolutional neural network (CNN). In the following, we describe the encoder-decoder architecture in more detail. For better readability, we omit weights and biases in the equations.

4.1.1 Encoder

The task of an encoder is to generate a summary of the whole source sentence f_1^J that can then be used by the decoder to create the target hypothesis. This is done in two steps:

1. Embed the one-hot vectors that are the network's input
2. Iterate over all embeddings using an LSTM and use its last state as the summary of the source sentence

The input to the network is a stream of one-hot vectors. Such a one-hot vector $\boldsymbol{f}_j$ has the size $|V_s|$ of the source vocabulary V_s and is defined as:

$$(\boldsymbol{f}_j)_n = \begin{cases} 1\,, & \text{if } n = z \\ 0\,, & \text{otherwise} \end{cases} \tag{4.1}$$

where z is the index of f_j in the vocabulary.

To reduce the size of the handled vectors and to provide information about the words, these one-hot vectors are multiplied to a trainable shared embedding matrix $\boldsymbol{W}_{enc}$. The resulting vector $\boldsymbol{W}_{enc}\boldsymbol{f}_j$ is a compact representation of the input word f_j that lies in the same shared vector space as all other embedded source words. Afterwards, an LSTM iterates over the output of the embedding layer one word at a time, accumulating a summary of the sentence. By repeatedly applying the following formula, the complete source sentence may be stored in $\boldsymbol{h}_J$.

$$\boldsymbol{h}_j = \mathrm{LSTM}(\boldsymbol{f}_j, \boldsymbol{h}_{j-1}) \tag{4.2}$$

4.1.2 Decoder

After the whole source sentence has been summarized by the encoder, the memory state $\boldsymbol{s}_0$ of the decoder LSTM is initialized based on the last state of the encoder LSTM, as written in Equation 4.3. The decoder LSTM is then used to predict the next target word. To this end, it receives the previous target word $\boldsymbol{e}_{i-1}$ as an input. Its state is used as an intermediate vector $\boldsymbol{t}_i$. It is transformed to a vector with the size $|V_t|$ of the target vocabulary V_t by another matrix $\boldsymbol{W}_{dec}$. In order to transform $\boldsymbol{t}_i$ to a normalized probability distribution that indicates how likely each word in the target vocabulary is to be the next target word, a softmax layer is applied. The relevant formulas are therefore:

$$\boldsymbol{s}_0 = \tanh\left(\boldsymbol{h}_J\right) \tag{4.3}$$

$$\boldsymbol{t}_i = \mathrm{LSTM}(\boldsymbol{e}_{i-1}, \boldsymbol{t}_{i-1}) \tag{4.4}$$

$$\Pr(\boldsymbol{e}_i | \boldsymbol{e}_1^{i-1}, \boldsymbol{f}_1^I) = \mathrm{softmax}(\boldsymbol{t}_i) \tag{4.5}$$

where the softmax function is defined as

$$\mathrm{softmax} : \mathbb{R}^K \to [0,1]^K \tag{4.6}$$

$$\mathrm{softmax}(\boldsymbol{x})_m = \frac{\exp(\boldsymbol{x}_m)}{\sum_{k=1}^{K} \exp(\boldsymbol{x}_k)} \tag{4.7}$$

The structure of the whole encoder-decoder network is depicted in Figure 4.1.

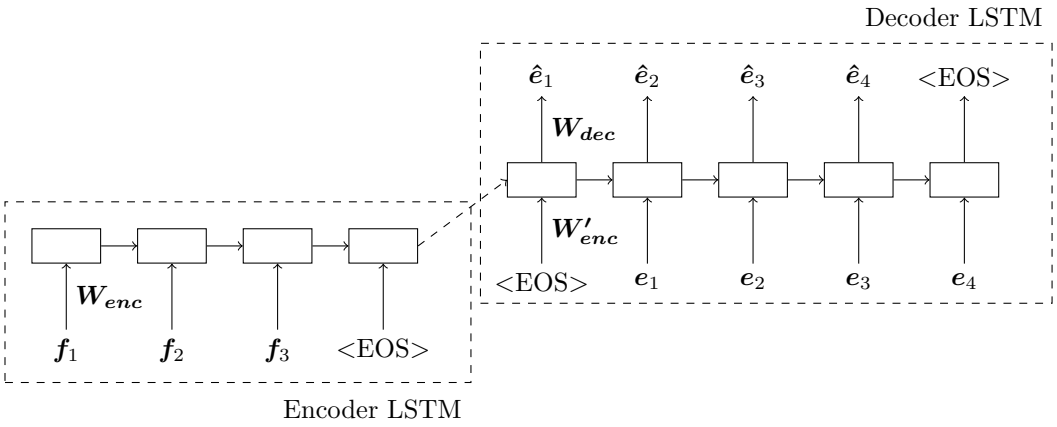


Figure 4.1: Structure of an encoder-decoder network. The input sentence is ($\boldsymbol{f}_1$, $\boldsymbol{f}_2$, $\boldsymbol{f}_3$), the generated output is ($\hat{\boldsymbol{e}}_1$, $\hat{\boldsymbol{e}}_2$, $\hat{\boldsymbol{e}}_3$, $\hat{\boldsymbol{e}}_4$). During training, the correct target words $\boldsymbol{e}_1^I$ are fed back to the decoder LSTM. Image adapted from [Sutskever et al., 2014].

4.2 Attention-based NMT

As depicted in Figure 4.1, using an encoder-decoder system requires that all the information of the source sentence f_1^J is encoded in a single fixed-sized vector $\boldsymbol{h}_J$. This is apparent from the Equations 4.2 to 4.4 as well. The only connections of the decoder LSTM to the source sentence is $\boldsymbol{h}_J$. If not all information is captured, the resulting prediction will not reflect the source correctly.

This poses an obvious problem. As the source sentence becomes longer, it contains more information. Because the size of $\boldsymbol{h}_J$ is fixed, the network becomes unable to encode the whole source sentence, and forgets parts of it. Bahdanau et al. [2014] show that the performance of an encoder-decoder system decreases drastically for long sequences.

As a remedy, they propose a different architecture. Instead of calculating a fixed source summary once at the beginning and using it for the whole decoding process, they calculate different context vectors $\boldsymbol{c}_i$ for every time step and feed it to the decoder LSTM. This enables the network to focus on different parts of the source sentence at each time step and avoids losing available information. Therefore, it behaves similar to human translators. Instead of first remembering the complete source sentence and then writing down the translation, they repeatedly look up different parts of the source.

In the following sections, we describe the modifications to the different model parts that are needed for attention.

4.2.1 Bidirectional Encoder

Instead of a single LSTM that processes the source words $\boldsymbol{f}_1^J$ from left to right, Bahdanau et al. [2014] propose to use an additional LSTM to iterate over $\boldsymbol{f}_1^J$ from $\boldsymbol{f}_J$ to $\boldsymbol{f}_1$. We write forward $\overrightarrow{\text{LSTM}}$ to denote the LSTM that iterates from $j = 1$ to $j = J$ and backward $\overleftarrow{\text{LSTM}}$ for the LSTM that iterates from $j = J$ to $j = 1$. The states of both LSTM cells are written as $\overrightarrow{\boldsymbol{h}}_j$ and $\overleftarrow{\boldsymbol{h}}_j$ accordingly.

$$\overrightarrow{\boldsymbol{h}}_j = \begin{cases} 0\,, & \text{if } j = 0 \\ \overrightarrow{\text{LSTM}}(\boldsymbol{f}_j, \overrightarrow{\boldsymbol{h}}_{j-1})\,, & \text{otherwise} \end{cases} \tag{4.8}$$

$$\overleftarrow{\boldsymbol{h}}_j = \begin{cases} 0\,, & \text{if } j = J+1 \\ \overleftarrow{\text{LSTM}}(\boldsymbol{f}_j, \overleftarrow{\boldsymbol{h}}_{j+1})\,, & \text{otherwise} \end{cases} \tag{4.9}$$

$$\boldsymbol{h}_j = \begin{bmatrix} \overrightarrow{\boldsymbol{h}}_j \\ \overleftarrow{\boldsymbol{h}}_j \end{bmatrix} \tag{4.10}$$

The concatenation of the states $\overrightarrow{\boldsymbol{h}}_j$ and $\overleftarrow{\boldsymbol{h}}_j$ corresponds to a summary of the whole source sentence with focus on the current source position j. It therefore provides a more complete context for $\boldsymbol{f}_j$ than could be gained using a one-directional encoder.

4.2.2 Attention Layer

The task of the attention layer is to compute a context vector $\boldsymbol{c}_i$ for every decoder time step i. It is a weighted average of all source representations $\boldsymbol{h}_1^J$ that have been created by the bidirectional encoder. The weighting algorithm allows the network to pick those source representations that are the most important for the next target word prediction. To compute the weighted average, the attention layer requires a decoder state $\boldsymbol{s}_{i-1}$. How this state is computed is explained in Section 4.2.3.

At every time step i, a scalar energy value $\tilde{e}_{j,i}$ is determined for each source representation $\boldsymbol{h}_j$.

$$\tilde{e}_{j,i} = \boldsymbol{v}^T \tanh(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j) \tag{4.11}$$

where $\boldsymbol{v}^T$ is a weight vector. Because this computation depends on the previous decoder state $\boldsymbol{s}_{i-1}$, $\tilde{e}_{j,i}$ can vary between different time steps. It indicates how much of $\boldsymbol{h}_j$ should be used for the next context vector $\boldsymbol{c}_i$, so a high energy implies a focus on the corresponding source representation.

The energies $\tilde{e}_{1,i}, \ldots, \tilde{e}_{J,i}$ are normalized using a softmax layer. The $\alpha_{1,i}, \ldots, \alpha_{J,i}$ are scalar weight values.

$$\alpha_{j,i} = \frac{\exp(\tilde{e}_{j,i})}{\sum_{j'=1}^{J} \exp(\tilde{e}_{j',i})} \tag{4.12}$$

Using these weights, the context vector $\boldsymbol{c}_i$ is computed as a weighted sum of all source representations $\boldsymbol{h}_1^J$.

$$\boldsymbol{c}_i = \sum_{j=1}^{J} \alpha_{j,i} \boldsymbol{h}_j \tag{4.13}$$

4.2.3 Decoder

Once the context vector $\boldsymbol{c}_i$ is determined, it is combined with the previous decoder state $\boldsymbol{s}_{i-1}$ and the previous target word $\boldsymbol{e}_{i-1}$ and fed through a *readout layer*. It computes the intermediate vector $\tilde{\boldsymbol{t}}_i$ as a linear combination of these three inputs.

$$\tilde{\boldsymbol{t}}_i = \boldsymbol{c}_i + \boldsymbol{s}_{i-1} + \boldsymbol{e}_{i-1} \tag{4.14}$$

where $\boldsymbol{W}_t$, $\boldsymbol{U}_t$ and $\boldsymbol{V}_t$ are weight matrices.

The output $\tilde{\boldsymbol{t}}_i$ is passed through a *maxout layer* as proposed by Goodfellow et al. [2013]. There, the values are separated in groups of size k, and only the largest value of each group is passed forward.

$$\boldsymbol{t}_i = \text{maxout}(\tilde{\boldsymbol{t}}_i) = \left[\max\left\{\tilde{\boldsymbol{t}}_{ilk}, \ldots, \tilde{\boldsymbol{t}}_{ilk+k-1}\right\}\right]_{l=0,\ldots,\frac{N}{k}}^{T} \tag{4.15}$$

where $\tilde{\boldsymbol{t}} \in \mathbb{R}^N$. Then, a softmax layer is applied to determine the most likely next target word $\boldsymbol{e}_i$.

$$\Pr(\boldsymbol{e}_i | \boldsymbol{e}_1^{i-1}, \boldsymbol{f}_1^J) = \text{softmax}(\boldsymbol{t}_i) \tag{4.16}$$

Finally, the new decoder state $\boldsymbol{s}_i$ is computed as the output of the decoder LSTM, given the target word $\boldsymbol{e}_i$ and te context vector $\boldsymbol{c}_i$

$$\boldsymbol{s}_i = \text{LSTM}([\boldsymbol{e}_i^T, \boldsymbol{c}_i^T]^T, \boldsymbol{s}_{i-1}) \tag{4.17}$$

Figure 4.2 depicts the flow of information in the attention-based NMT system. The weighted average of all source word representations is updated at each time step, resulting in different context vectors for each target word.

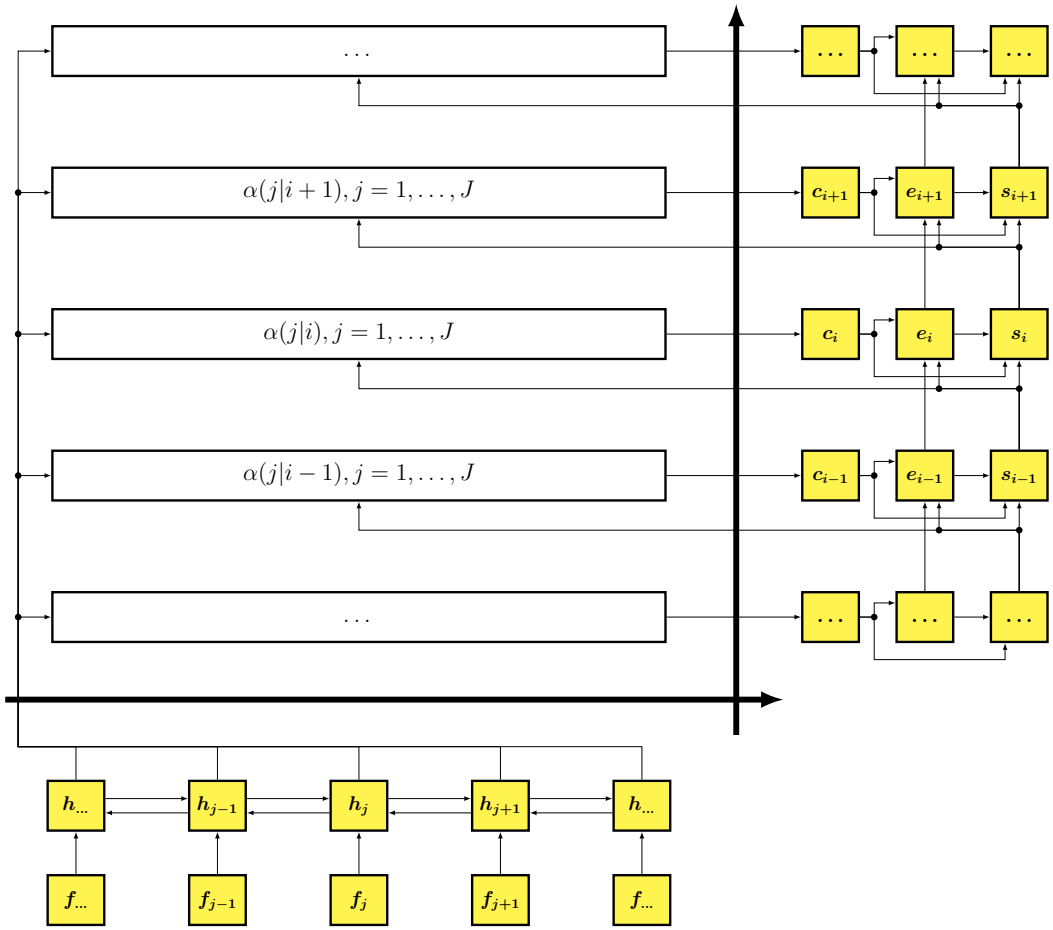


Figure 4.2: The structure of a network using attention. The bidirectional encoder and the decoder are marked yellow. At each time step, the context vector $\boldsymbol{c}_i$ is recomputed using a weighted sum of all source representations $\boldsymbol{h}_1^J$.

Chapter 5

Extensions of the Attention Mechanism

In Equation 4.11, one can see that the attention energy of a given source position j is not used to determine how much attention energy should be given to a different source position j' with $j' \neq j$. The determined energies do influence each other in the normalizing step (see Equation 4.12). However, this only results in a minor dependency. It is not possible to attend to a specific source position only because attention is payed to another position as well.

We construct and evaluate extensions of the usual neural network (NN) structure that do not suffer from said shortcoming. For this, we add dependencies to the computation of the context vector that allow the network to take into account how much it attended to other source positions.

A theoretic example for a situation where this might be beneficial is the sentence "The house is big" that should be translated into German. The first target word is likely dependent upon the first source word, so attention is paid to "The". As German has three different articles ("der", "die" and "das"), the correct one has to be chosen based on the following noun. If the NN can access previous attention, it may have learned this dependency and decide to also attend to the noun "house", resulting in the decision to use the neuter article "das". In the similar sentence "Whose house is big?", the correct translation of "Whose" ("Wessen") does not depend upon the following noun. The NN can notice this and does not need to attend to it. Because the usual attention mechanism attends to all words in parallel, it cannot leverage this knowledge. It always has to attend to the noun as well, in case the previous source word requires it.

Additionally, we enable the attention layer to save information across time steps. This may theoretically be helpful for the translation of the French sentence "Je ne veux pas arrêter" into the English sentence "I do not want to stop". The French "ne pas" usually encapsulates the verb that is negated. Knowing that at the last decoding time step attention was payed to "ne ... pas" could make it easier for the model to focus on the correct verb. This may prevent it from attending to the wrong one and inadvertently translating the sentence as "I do not stop to want".

A third positive aspect of the extended attention layer may be the ability to avoid *over-* and *under-translations*. They occur if the network translates the same

parts of the source sentence multiple times or omits them completely. By storing the knowledge of which source words have been attended to, the network may learn the concept of *fertility*. It would then try to attend to all source positions at least once, but not multiple times, thus avoiding over- and under-translations.

Some of our modifications are based upon an unpublished paper by Zhang et al. [2016]. They propose to remove the attention mechanism that is described by Bahdanau et al. [2014] and replace it with a layer that iterates over the source representations. This technique is explained in Section 5.2. Then, we extend their work with various architectures and dependencies. We apply both long short-term memory (LSTM) and gated recurrent units (GRUs) as the attention layer and generalize the architecture to two-dimensional LSTM (2DLSTM) attention. Finally, in Section 5.4, we propose a novel architecture that only uses a single 2DLSTM and no explicit encoder or decoder.

5.1 Recalculating the Encoder State

In the vanilla encoder-decoder attention architecture, the source sentence is encoded once at the beginning of the translation, as described in Section 4.2. This evidently implies that the source representation is independent from the decoder state and not updated to reflect the current process of the translation.

We propose to make the source representation dependent on the decoder state. To achieve this, all $\boldsymbol{h}_j$ are recomputed at every decoding time step i. The rest of the network remains the same as in the encoder-decoder architecture with attention. Thus, the equations for the encoding layer (formerly Equations 4.8 to 4.10) can be expressed as:

$$\overrightarrow{\boldsymbol{h}}_{j,i} = \begin{cases} 0\,, & \text{if } j = 0 \\ \overrightarrow{RNN}([\boldsymbol{f}_j^T; \boldsymbol{s}_{i-1}^T]^T, \overrightarrow{\boldsymbol{h}}_{j-1,i})\,, & \text{otherwise} \end{cases} \tag{5.1}$$

$$\overleftarrow{\boldsymbol{h}}_{j,i} = \begin{cases} 0\,, & \text{if } j = J+1 \\ \overleftarrow{RNN}([\boldsymbol{f}_j^T; \boldsymbol{s}_{i-1}^T]^T, \overleftarrow{\boldsymbol{h}}_{j+1,i})\,, & \text{otherwise} \end{cases} \tag{5.2}$$

$$\boldsymbol{h}_{j,i} = \begin{bmatrix} \overrightarrow{\boldsymbol{h}}_{j,i} \\ \overleftarrow{\boldsymbol{h}}_{j,i} \end{bmatrix} \tag{5.3}$$

5.2 One-Dimensional (1D) Attention

Zhang et al. [2016] propose to use a GRU layer to replace the usual attention mechanism. Instead of predicting attention weights that are then used to compute a weighted sum of all source representations, the GRU should summarize the

necessary parts of the source sentence directly. The following explanations are formulated in the more general form of an recurrent neural network (RNN). All the concepts can easily be transferred to different recurrent setups. In our experiments, we evaluate the performance of both GRU and LSTM, respectively.

The attention mechanism as described in [Bahdanau et al., 2014] computes the energies based on the previous decoder state and the source representations. This is shown in Equation 4.11. Zhang et al. [2016] initialize the state $\boldsymbol{a}_{0,i}$ of an RNN with the decoder state. They argue that the RNN can use this information to assess which source words are important at the current time step and which are not. It does so by controlling how much of the current source word is added to the RNN state. Therefore, they use the RNN to iterate over all source representations $\boldsymbol{h}_1^J$. Once all source words have been processed, either the final state $\boldsymbol{a}_{J,i}$ or a combination of multiple states $\boldsymbol{a}_{j,i}$ is used as the context vector $\boldsymbol{c}_i$. If the last state is used, this leads to the following equations.

$$\boldsymbol{a}_{0,i} = \tanh(\boldsymbol{s}_{i-1}) \tag{5.4}$$

$$\boldsymbol{a}_{j,i} = \mathrm{RNN}(\boldsymbol{h}_j, \boldsymbol{a}_{j-1,i}) \tag{5.5}$$

$$\boldsymbol{c}_i = \boldsymbol{a}_{J,i} \tag{5.6}$$

Alternatively, Zhang et al. [2016] compute $\boldsymbol{c}_i$ as the average of all $\boldsymbol{a}_{1,i}, \ldots, \boldsymbol{a}_{J,i}$.

$$\boldsymbol{c}_i = \frac{1}{J} \sum_{j=1}^{J} \boldsymbol{a}_{j,i} \tag{5.7}$$

As explained in the introduction, the network might benefit from storing information about the attention at time step i, in order to use it at a later time step i' with $i' > i$. This can be achieved by storing it in the context vector $\boldsymbol{c}_i$. By copying the information to the decoder state $\boldsymbol{s}_i$, the network will be able to access the stored knowledge in the attention layer at time step $i + 1$. A visualization of such an extended attention layer using an LSTM is shown in Figure 5.1.

Using the target word $\boldsymbol{e}_{i-1}$ instead of the decoder state $\boldsymbol{s}_{i-1}$ to initialize the RNN results in a bad performance. In such a model, the attention layer does not know about the complete target history and can therefore not correctly determine the context vector.

5.2.1 Additional Attention Layer

Zhang et al. [2016] compare both using the last state $\boldsymbol{a}_{J,i}$ and using the average of all states $\frac{1}{J}\sum_{j=1}^{J} \boldsymbol{a}_{j,i}$ as the context vector $\boldsymbol{c}_i$. They come to the conclusion that the former is the superior configuration.

In order to verify their findings, we generalize the technique of computing the average of all RNN states to a weighted sum. Given the decoder state $\boldsymbol{s}_{i-1}$ and

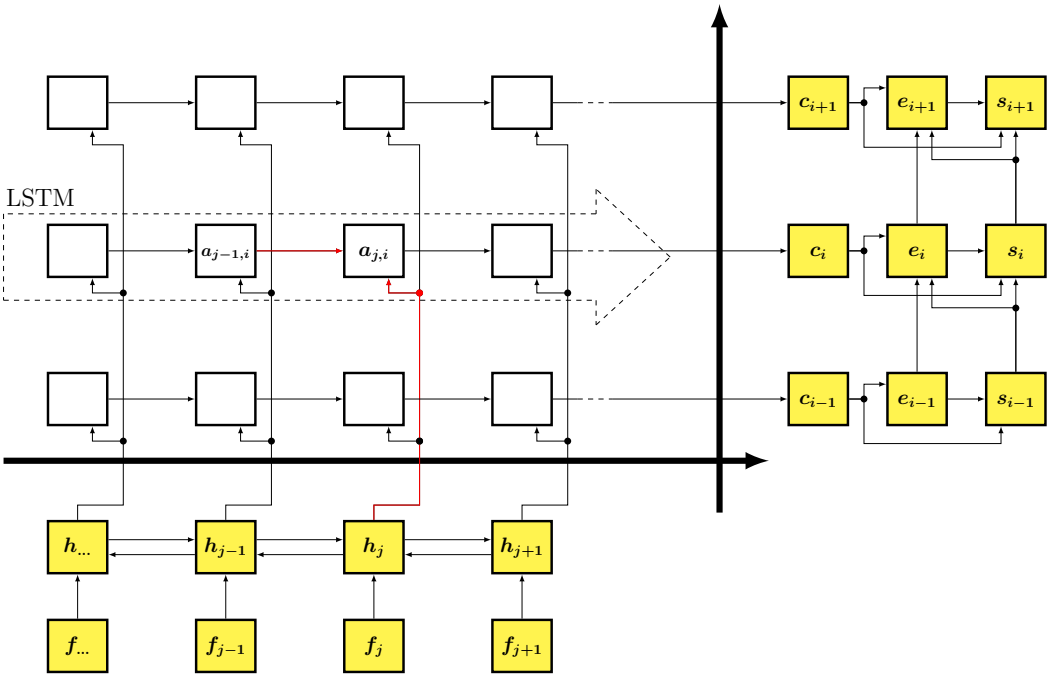


Figure 5.1: 1D attention using an LSTM. The encoder and decoder are highlighted yellow. At time step i, one LSTM iterates over all source representations $\boldsymbol{h}_1^J$. The final state $\boldsymbol{a}_{J,i}$ is used as the context vector $\boldsymbol{c}_i$. The initialization of the LSTM based on the decoder state $\boldsymbol{s}_{i-1}$ is omitted for simplicity. The flow of information at time step (j, i) is highlighted in red.

the RNN states $\boldsymbol{a}_{1,i}, \ldots, \boldsymbol{a}_{J,i}$, one can compute a weighted average analogue to the attention mechanism by Bahdanau et al. [2014] in Equations 4.11 to 4.13:

$$\tilde{e}_{j,i} = \boldsymbol{v}^T \tanh\left([\boldsymbol{s}_{i-1}^T; \boldsymbol{a}_{j,i}^T]^T\right) \tag{5.8}$$

$$\alpha_{j,i} = \frac{\exp(\tilde{e}_{j,i})}{\sum_{j'=1}^{J} \exp(\tilde{e}_{j',i})} \tag{5.9}$$

$$\boldsymbol{c}_i = \sum_{j=1}^{J} \alpha_{j,i} \boldsymbol{a}_{j,i} \tag{5.10}$$

where $\boldsymbol{v}^T$ is a weight vector. The simple arithmetic mean of all RNN states is a special case of this. It can be simulated by the network by outputting the same $\tilde{e}_{j,i}$ regardless of $\boldsymbol{s}_{i-1}$ or $\boldsymbol{a}_{j,i}$.

5.2.2 Providing the Decoder State

The RNN state is initialized with the decoder state in order to provide the target history. Alternatively, either the decoder state or the last target word could be appended to the RNN's input. This would still allow the network to use the target information within the attention layer. When the decoder state is appended, the Equations 5.4 and 5.5 are replaced by

$$\boldsymbol{a}_{0,i} = \boldsymbol{z} \tag{5.11}$$

$$\boldsymbol{a}_{j,i} = \mathrm{RNN}([\boldsymbol{h}_j^T; \boldsymbol{s}_{i-1}^T]^T, \boldsymbol{a}_{j-1,i}) \tag{5.12}$$

where $\boldsymbol{z}$ can either be a trainable parameter, or a vector of zeros.

5.3 Two-Dimensional (2D) Attention

In Section 5.2 we show that the network is able to remember its attention from previous decoding time steps by storing it in the context vector and copying it to the decoder state. However, this process is not only complex and therefore potentially difficult for the network to learn and leverage. The network also requires the decoder state $\boldsymbol{s}_{i-1}$ to contain both the target history and knowledge about the previous attention distribution. This might be harmful during the actual prediction of the next target word as described in Equations 4.14 and 4.17.

In addition to this observation, we notice that during the computation of the context vector, all the needed information is available if the whole source sentence, and all previously generated target words are given. This gave rise to the idea to use a 2DLSTM to process these two sequences.

We arbitrarily decide the dimension that depends on the source sentence to be the first dimension. The second dimension therefore depends on the target sequence. This choice does not influence the training. Defining the purpose of each dimension will simplify the following discussions. By iterating in the direction of the first dimension, the 2DLSTM learns about all source words. The information about the previously hypothesized target words is supplied by the second dimension. Within the 2DLSTM, the current status of the computation has to be named by 2D indices (j, i). At each position (j, i), we provide both the source representation $\boldsymbol{h}_j$ and the previous target word $\boldsymbol{e}_{i-1}$ as the input to the 2DLSTM. This can be done by concatenating them. Our hypothesis is that after processing the whole source and the target history, ie. at position $(J, i-1)$, the 2DLSTM is able to provide the context vector $\boldsymbol{c}_i$ that will be used to predict the next target word.

The whole setup of 2DLSTM is shown in Figure 5.2. Formally, it is defined as

$$\boldsymbol{a}_{j,i} = \mathrm{2DLSTM}([\boldsymbol{h}_j^T; \boldsymbol{e}_{i-1}^T]^T, \boldsymbol{a}_{j-1,i}, \boldsymbol{a}_{j,i-1}) \tag{5.13}$$

$$\boldsymbol{c}_i = \boldsymbol{a}_{J,i} \tag{5.14}$$

For a naive implementation, there would be one major problem. At each time step i, the 2DLSTM should iterate over all previously generated target words $\boldsymbol{e}_1^{i-1}$. Since at $i = 1$ no target history is available yet, the second dimension has a length of 0. This would make it impossible to iterate over the source sentence for the first time and no context vector $\boldsymbol{c}_1$ could be computed. Therefore, the network would be prevented from starting the translation.

We solve this problem by adding an additional <BOS> token, that represents the sentence-start, to the beginning of the target sentence. This was only done during the computation of the 2DLSTM and did not influence the encoder or decoder of the network. Because, in the context of the 2DLSTM, every target sentence starts with the special token <BOS>, the aforementioned problem is solved. At $i = 1$, the sentence-start token is already given, thus the second dimension is of length 1. Theoretically, the <BOS> token could be removed once the first target word has been hypothesized. However, this could increase the complexity, since the network would then have to differentiate between <BOS> and a real hypothesized target word as the first input. We therefore append the <BOS> token at every decoding step for the 2DLSTM attention setup.

As an alternative to appending the target word $\boldsymbol{e}_{i-1}$ to the input, one could use the decoder state $\boldsymbol{s}_{i-1}$ as well. However, the decoder states are not known before the training process starts. Appending the decoder states to the input would therefore prevent the optimization described in the following section.

5.3.1 Difference between Training and Decoding

As explained above, at each time step i, the 2DLSTM has to iterate over both the whole sequence of source sentences, as well as all previously hypothesized target words. Because the number of computations for J source and I target words is $\mathcal{O}(J \cdot I)$ for a single time step and $\mathcal{O}(J \cdot I^2)$ for all I time steps, optimizing this process is quite important.

During the training process, all target words are already known beforehand. For the computation of the 2DLSTM cells, the knowledge of the correct target words can be used. Before starting the training process for any i, a single 2DLSTM can iterate over both the complete source and target sentence. When the 2DLSTM states are stored in memory, the need for further computations during the training step is eliminated. Once, for a given time step i, the 2DLSTM states are needed, they can be read from memory, yielding a time complexity of $\mathcal{O}(J \cdot I)$ for all I time steps. As shown by Voigtlaender et al. [2016], all states of a 2DLSTM located in one diagonal can be computed in parallel (see Section 3.3.2). Therefore, the time complexity can be reduced to $\mathcal{O}(J + I)$.

Unfortunately, this is not possible during decoding. Because the target sentence is not known in advance, only those target words can be used for the 2DLSTM that

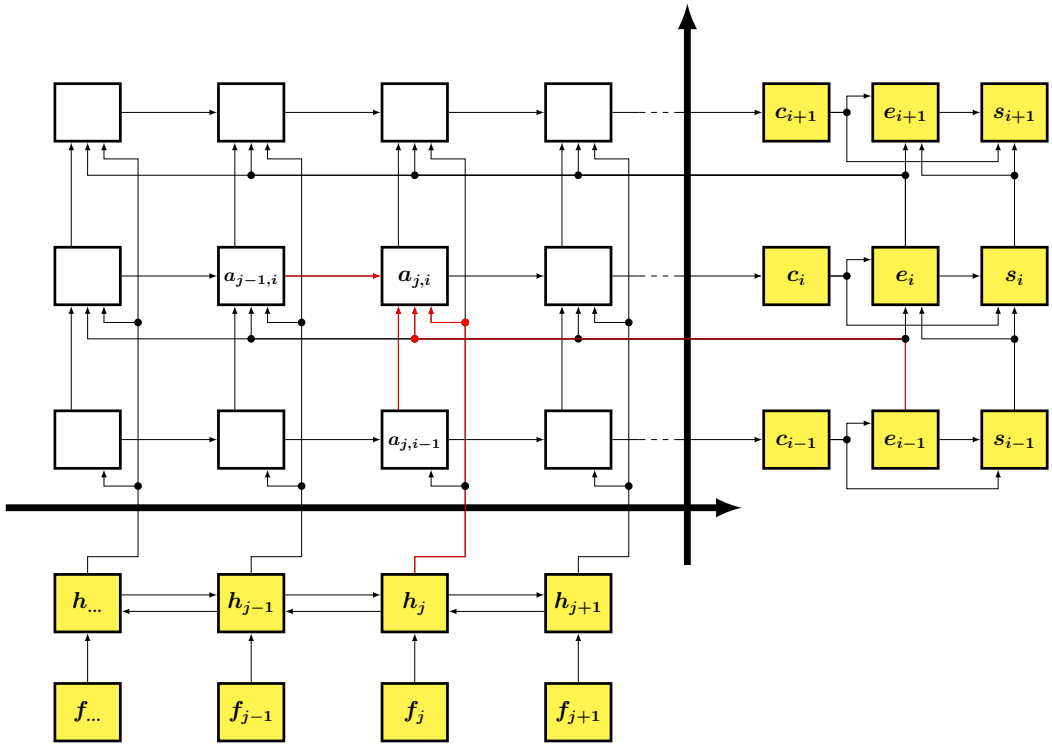


Figure 5.2: The setup of a 2DLSTM attention model. The encoder and decoder are marked yellow. At time step (j, i), the 2DLSTM receives the states from the time steps $(j-1, i)$ and $(j, i-1)$, as well as the concatenation of $\boldsymbol{h}_j$ and $\boldsymbol{e}_{i-1}$. The final 2DLSTM state at (J, i) is used as the context vector $\boldsymbol{c}_i$. The flow of information at time step (j, i) is highlighted in red.

have already been hypothesized. However, an optimal implementation could still leverage the fact that for a target sequence $\boldsymbol{e}_1^i$, the history $\boldsymbol{e}_1^{i-1}$ has already been processed in the past. To extend the 2DLSTM to the new target word, only one additional row has to be computed, as shown in Figure 5.3. The cell states $\boldsymbol{a}_{j,i-1}$ that this row depends upon can be read from the memory of the decoding time step $i-1$.

This was not possible with the implementation in the RWTH extensible training framework for universal recurrent neural networks (RETURNN) [Doetsch et al., 2016, Voigtlaender et al., 2016] which we use. We were therefore forced to recompute the 2DLSTM from scratch at every decoding time step, resulting in $\mathcal{O}(I \cdot (I+J))$ computations for the 2DLSTM per translated sentence.

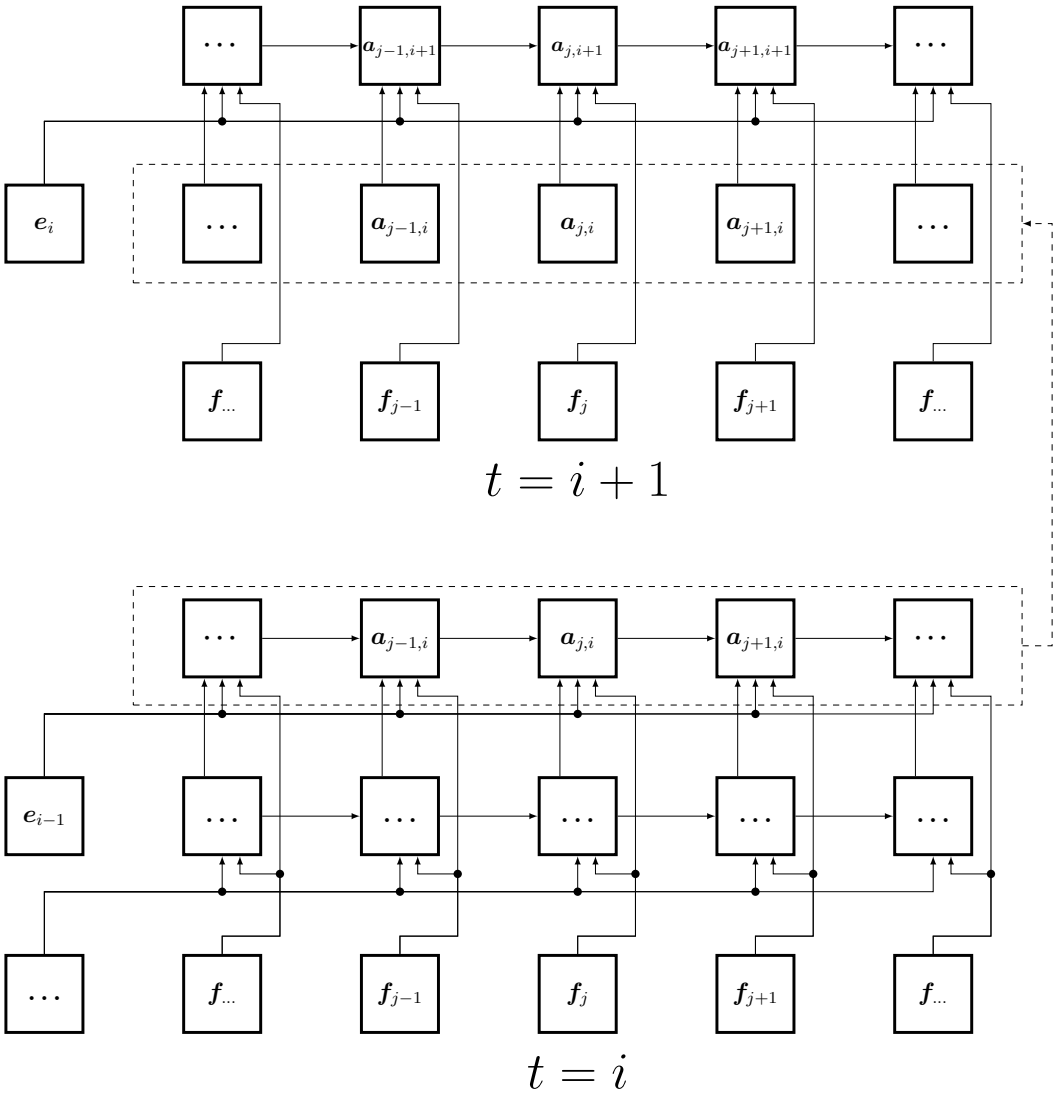


Figure 5.3: Reusing the 2DLSTM during decoding. By storing the top row of the 2DLSTM in memory, one does not have to recompute the whole 2DLSTM during the next timestep.

5.3.2 Optimization of the Backward Pass

To make the training fast, it is important to reduce the number of necessary computations as much as possible. Especially matrix-vector multiplications and sigmoid functions can become very time consuming for large setups.

In the formulas of the gradient of the 2DLSTM described in Appendix A.1.3, one can observe that the gate values during the forward pass can be reused. This can be leveraged eg. in Equation A.33, where being able to access the output gate value $\boldsymbol{o}_{t,t'}$ that was calculated during the forward pass eliminates the need to recompute $\sigma(\boldsymbol{M}_o[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t,t'}^T]^T)$. In the implementation in RETURNN by Voigtlaender et al. [2016], only the gate values $\boldsymbol{i}_{t,t'}$, $\boldsymbol{f}_{t,t'}$, $\boldsymbol{\lambda}_{t,t'}$ and $\boldsymbol{o}_{t,t'}$, as well as the LSTM state $\boldsymbol{c}_{t,t'}$ are saved. The input candidate $\tilde{\boldsymbol{c}}_{t,t'}$ that is needed in Equations A.38 and A.40 and the value of $\tanh(\boldsymbol{c}_{t,t'})$, used in Equations A.31 and A.32, are not stored. Instead, they are recomputed based on the available information. Using Equation 3.32 for $\boldsymbol{o}_{t,t'} \in \mathbb{R}^n$

$$\boldsymbol{h}_{t,t'} = \boldsymbol{o}_{t,t'} \circ \tanh(\boldsymbol{c}_{t,t'}) \tag{5.15}$$

Voigtlaender et al. [2016] define

$$(\tanh(\boldsymbol{c}_{t,t'}))_k = \begin{cases} (\boldsymbol{h}_{t,t'})_k / (\boldsymbol{o}_{t,t'})_k\,, & \text{if } (\boldsymbol{o}_{t,t'})_k \neq 0 \\ 0\,, & \text{otherwise} \end{cases} \forall k \in \{1, \ldots, n\} \tag{5.16}$$

This avoids the expensive computation of $\tanh(\boldsymbol{c}_{t,t'})$ based on $\boldsymbol{c}_{t,t'}$. Similarly, using Equation 3.30 for $\boldsymbol{c}_{t,t'} \in \mathbb{R}^n$

$$\boldsymbol{c}_{t,t'} = \boldsymbol{f}_{t,t'} \circ (\boldsymbol{\lambda}_{t,t'} \circ \boldsymbol{c}_{t-1,t'} + (\boldsymbol{1} - \boldsymbol{\lambda}_{t,t'}) \circ \boldsymbol{c}_{t,t'-1}) + \boldsymbol{i}_{t,t'} \circ \tilde{\boldsymbol{c}}_{t,t'} \tag{5.17}$$

they define

$$(\tilde{\boldsymbol{c}}_{t,t'})_k = \begin{cases} (\boldsymbol{c}_{t,t'})_k - (\boldsymbol{f}_{t,t'})_k \Big((\boldsymbol{\lambda}_{t,t'})_k (\boldsymbol{c}_{t-1,t'})_k \\ \quad + (1 - (\boldsymbol{\lambda}_{t,t'})_k)(\boldsymbol{c}_{t,t'-1})_k \Big) / (\boldsymbol{i}_{t,t'})_k\,, & \text{if } (\boldsymbol{i}_{t,t'})_k \neq 0 \\ 0\,, & \text{otherwise} \end{cases} \forall k \in \{1, \ldots, n\} \tag{5.18}$$

We have noticed that due to the numerical instability of floating point operations, this causes the values of $\tanh(\boldsymbol{c}_{t,t'})$ and $\tilde{\boldsymbol{c}}_{t,t'}$ during the backpropagation to be slightly different from the forward pass. However, based on the fact that the absolute error between both values is less than 10^{-9}, it does not significantly influence the training process. We therefore run our experiments using the original implementation.

5.4 2D Sequence to Sequence (2D seq2seq) Model

In the previous sections, we have explained two possibilities to improve the currently used encoder-attention-decoder setup by replacing the attention mechanism. In this section, we propose a different architecture that does not have an LSTM encoder and decoder.

In the new architecture, a 2DLSTM, as it is used for the 2D attention in Section 5.3, processes all source words and the target history. We have come up with an idea where we omit all other parts of the network, and use only a single 2DLSTM to generate the hypotheses. The 2DLSTM can be interpreted to provide a 2D mapping of the source and target words to a shared space. The resulting network has the following structure:

1. An embedding matrix transforms each one-hot vector into a shared vector space
2. The 2DLSTM iterates over all source embeddings and all target history embeddings
3. The last horizontal 2DLSTM state $\boldsymbol{a}_{J,i}$ is transformed using a second embedding matrix
4. The obtained vector is normalized using a softmax layer and used as the probability distribution over the target words

This setup is described by the equations as follows:

$$\boldsymbol{a}_{j,i} = \text{2DLSTM}([\boldsymbol{f}_j^T; \boldsymbol{e}_{i-1}^T]^T, \boldsymbol{a}_{j-1,i}, \boldsymbol{a}_{j,i-1}) \tag{5.19}$$

$$\boldsymbol{e}_i = \text{softmax}(\boldsymbol{a}_{J,i}) \tag{5.20}$$

The 2D seq2seq architecture can be seen in Figure 5.4.

5.4.1 2D Encoder

Since the network reads the source sentence from left to right, it has an obvious drawback compared to models with a bidirectional encoder where the entire context is encoded in each position j. We not only try to add such an encoder to the network (still without using any special attention or decoder structure), but also attempt to replace the encoder by another 2D network.

To achieve this, we use the sequence of source words for the first dimension and the inverted source sequence for the second dimension. After finishing the 2DLSTM computation, we use the elements $\boldsymbol{h}_{j,j}$ as our source representations $\boldsymbol{h}_j$. In other words, we take the major diagonal of the encoder matrix as the sequence of source

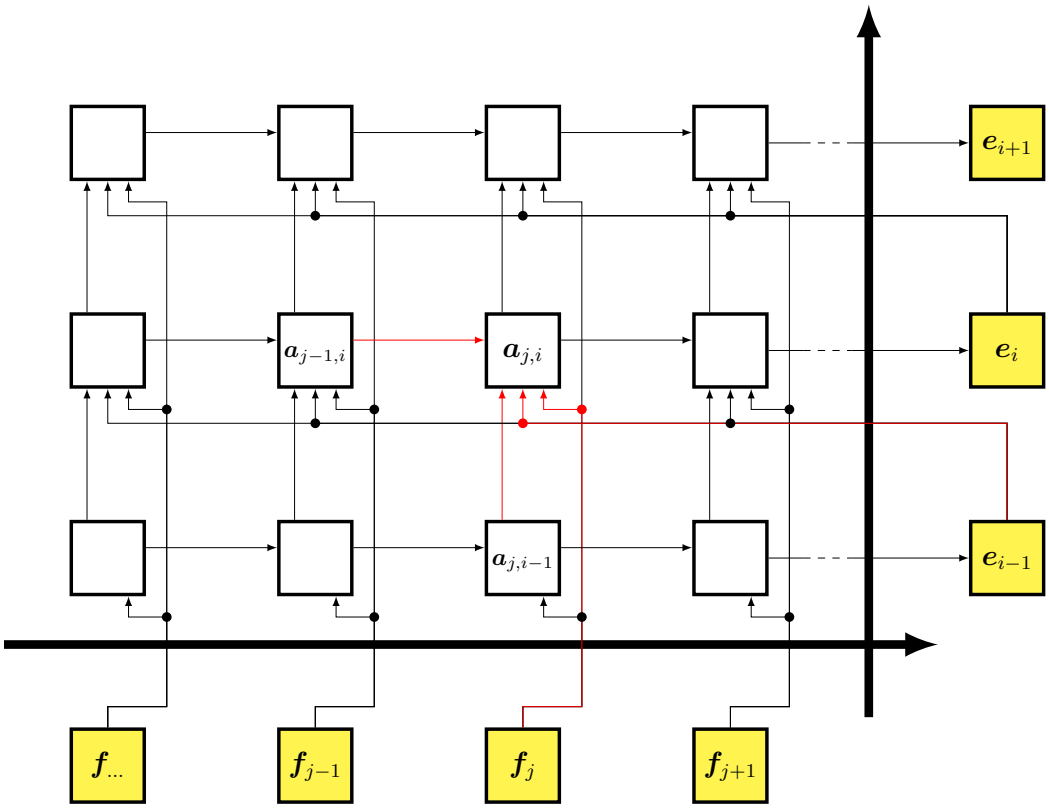


Figure 5.4: A 2D seq2seq architecture. The source sentence is embedded and directly fed to the 2DLSTM. Its final states $\boldsymbol{a}_{J,i}$ are used to predict the next target word. There is no explicit decoder state, the 2DLSTM keeps track of the target history internally. The flow of information at time step (j, i) is highlighted in red.

representations. As can be seen in Figure 5.5, for the elements at this diagonal, the network has seen the complete source sentence with focus on the individual word $\boldsymbol{f}_j$.

5.4.2 Weighting Mechanism

In addition to the 2D encoder, we experiment with applying a weighting mechanism before predicting the next target word. It uses the 2DLSTM state $\boldsymbol{a}_{J,i}$ to weight

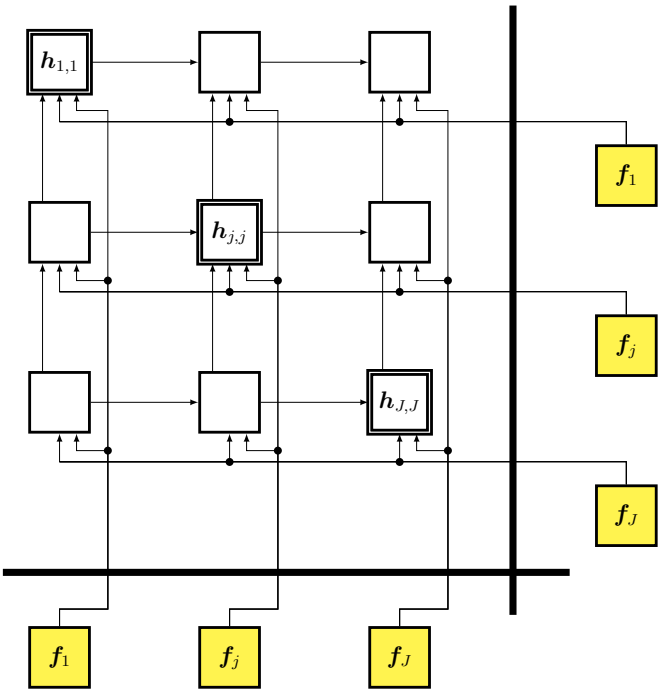


Figure 5.5: A 2D encoder. The marked states in the diagonal are used as the encodings $\boldsymbol{h}_j$. At those points, the network has seen the complete source sentence, the word at position j being the last one.

all states $\boldsymbol{a}_{1,i}, \ldots, \boldsymbol{a}_{J,i}$. This is done analogue to the weighting mechanism in the attention layer described in [Bahdanau et al., 2014].

$$\boldsymbol{a}_{j,i} = \text{2DLSTM}([\boldsymbol{f}_j^T; \boldsymbol{e}_{i-1}^T]^T, \boldsymbol{a}_{j-1,i}, \boldsymbol{a}_{j,i-1}) \tag{5.21}$$

$$\tilde{e}_{j,i} = \boldsymbol{v}^T \tanh\left([\boldsymbol{a}_{J,j}^T; \boldsymbol{a}_{j,i}^T]^T\right) \tag{5.22}$$

$$\alpha_{j,i} = \frac{\exp(\tilde{e}_{j,i})}{\sum_{j'=1}^{J} \exp(\tilde{e}_{j',i})} \tag{5.23}$$

$$\boldsymbol{t}_i = \sum_{j=1}^{J} \alpha_{j,i} \boldsymbol{a}_{j,i} \tag{5.24}$$

$$\boldsymbol{e}_i = \text{softmax}(\boldsymbol{t}_i) \tag{5.25}$$

where $\boldsymbol{v}^T$ is a weight vector.

Chapter 6

Experiments

To test the performance of our proposed modifications, we have performed a number of experiments. They were designed to show the influence of different variations of similar setups and to allow a judgment of which modifications are the most promising ones.

6.1 Preprocessing

In theory, a sufficiently complex artificial neural network (ANN) should be able to process any input in any format, as long as it is consistent. However, certain modifications to the input can simplify the data processing, thus reducing the need for complexity and making the network easier to train. This section presents three commonly used techniques that can be applied offline, ie. as a preprocessing step before the training.

6.1.1 Tokenization

Usually, the natural separation of words by spaces is used to split the input string into symbols, potentially followed by further splitting (see Section 6.1.2). However, simple space based splitting will cause problems due to punctuations: eg. question marks at the end of the sentence or commas in between would lead to unnecessarily duplicated vocabulary entries like “speak” and “speak?” if “speak” appears once in the middle of a sentence and once at the end of a question.

To avoid this issue, additional spacing is introduced in front of all punctuation, thus separating them from any connected word. The punctuation signs can then be processed like regular words.

6.1.2 Subword Units

The input to the neural network is a one-hot vector indicating the word of the source vocabulary and the output is a probability distribution over all words in the target vocabulary. This restricts the network to operate on a fixed set of words. Hence, all out of vocabulary (OOV) words are mapped to a special token (UNK).

Unfortunately, multiple problems arise if the source vocabulary has to be fixed to a specific size: firstly, there is no fixed list of possible named entities, so the test set is likely to contain some unknown words. Secondly, inflections can create many small variations of the same word, further inflating the size of a potential vocabulary. And thirdly: some languages like German use compounds like "Türklinke" ("door handle") that consist of individual words such as "Tür" ("door") and "Klinke" ("handle"). By using this technique, infinitely many new words can be created. It is clear that no network can be able to recognize and generate all these words. There are multiple possible solutions to this problem.

1. Copy all unknown input words verbatim to the target sentence. This can solve problems with named entities, as they can usually be copied from the source. But for unknown words other than named entities, it will copy them as well, leading to a translation that is full of words in the source language.

2. Use an additional dictionary for unknown words. Because it may contain words which are not known to the neural network (NN), this will reduce the rate of untranslatable words. However, it will not catch all of them.

3. Split unknown words into smaller, known parts. This technique is currently used most often.

Sennrich et al. [2015] have been the first to apply the byte pair encoding (BPE) compression algorithm, as described in [Gage, 1994], to source and target sequences in order to allow the NN to work with unknown source and target words. It splits unknown words in smaller, known chunks, thus enabling the network to improvise eg. compounds like "Türklinke" by concatenating the two known words "Tür" and "Klinke". Names on the other hand might be split on character level and copied one by one to the target sentence, i.e. the network is still capable of copying text.

The algorithm splits all words in the corpus on character level. The characters are treated as symbols. Then, at each iteration, the two symbols that occur consecutively the most often are merged and treated as one single new symbol during the next iterations. The information that these two symbols were merged is appended to a list that saves all merging operations. A predefined number of merging operations is the stop criterion.

Before unseen sentences are used in the decoding phase, all words in this test corpus are split on character level as well. Then, the previously generated list of merging operations is applied in the same order the operations have been saved. This generates a corpus that only consists of (sub-) words that are part of the network's vocabulary.

An example of this process can be seen in Table 6.1. Sennrich et al. [2015] note that completely unknown characters in the test set will still lead to OOV words.

Table 6.1: Example of the BPE algorithm. The compound word "airplane" is split into "air·p·l·an·e", potentially simplifying its translation because "air" is identified to be part of it.

corpus	symbols merged
a·n a·i·r·p·l·a·n·e i·n t·h·e a·i·r	a·n → an
an a·i·r·p·l·an·e i·n t·h·e a·i·r	a·i → ai
an ai·r·p·l·an·e i·n t·h·e ai·r	ai ·r → air
an air·p·l·an·e i·n t·h·e air	

6.1.3 Category Replacement

Besides usual words, text may also contain symbols from special categories, such as numbers or URLs. Both should usually be copied verbatim. While numbers could be split to individual digits by means of BPE and only increase the sentence length, URLs pose a bigger problem.

During the tokenization, an URL like "https://example.com/about-us" would be transformed to "https : / / example . com / about - us". A network trained for English to German translation would most likely generate "https : / / beispiel . com / über - uns", thus an unrelated URL. To avoid this problem, numbers as well as URLs can be replaced by a special placeholder in a preprocessing step. After the network has generated the hypothesis, this placeholder can then be replaced by the original URL or number. If there are more than one number or URL existing in the source and target sentence, the attention weights computed by the network (see Section 4.2) can be used to identify which target placeholder is aligned to which source word. Although category replacement is beneficial in statistical machine translation (SMT), we have observed that URLs occur seldom enough not to be important and that BPE is capable of splitting numbers to frequent n-gram sequences. Therefore, we do not use category replacement in our experiments.

6.2 Setup

In this section, we define the general setup of our experiments. Because many of the experiments take a long time to train, it has not been possible to redo them, once better hyperparameters such as the learning rate or model size have been found. Therefore, specific details are not necessarily the same across all experiments. These are individually highlighted. However, we attempt to make each comparison as fair as possible.

We train the models on two different corpora:

1. The workshop on machine translation (WMT) corpus (see Appendix A.3.1) is used for both German→English and English→German translation. It consists of the corpora *Europarl-v7*, *News-Commentary-v10* and *Common-Crawl*[1]. *newstest2015* is our development set, *newstest2016* and *newstest2017* are our test sets.

2. The international workshop on spoken language translation (IWSLT) indomain technology, entertainment, design (TED) talks training corpus (see Appendix A.3.2) is used to translate German→English.

We apply true-casing and BPE (see Section 6.1.2) with 20k merge operations. All of our experiments have the following setup:

1. Unless otherwise noted, the embedding matrices have the size of the encoder and decoder respectively.

2. During the training, each batch consists of 50 sentences, each no longer than 50 subword units.

3. The decoder long short-term memory (LSTM) is initialized with the last state of the bidirectional encoder LSTM.

4. The parameters are initialized using a xavier distribution (see Section 3.4.2).

5. As the optimization technique, the Adam optimizer (see Section 3.4.2) is used.

6. For the encoder-decoder architecture with the usual attention mechanism, as well as one-dimensional (1D) LSTM attention, the LSTM cells have additional peephole connections. For the two-dimensional LSTM (2DLSTM), these were omitted.

7. For decoding (see Section 2.2), we use a beam size of 12.

As our baseline, we use an encoder-decoder architecture with attention (see Section 4.2). It consists of a bidirectional encoder, an additive attention mechanism and a unidirectional decoder. The attention layer consists of a tanh nonlinearity followed by a softmax function. In the decoder, a maxout and a softmax layer are applied to determine the next prediction. The encoder and decoder recurrent neural networks (RNNs) are LSTMs. The sizes of the encoder, attention layer and decoder vary, they are specified for each experiment. Generally, the attention layer computes the weights based on source representations with the size of the encoder. However, it applies these weights to the representations with twice the

[1]For some German→English experiments, we add the newstest sets form 2008-2014 four times to the training corpus

Table 6.2: Evaluation of recomputing the encoding. Trained on WMT 2017 German→English, with an encoder/attention/decoder size of 500.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	26.4	31.2	27.1	53.6	48.9	53.3	12.5	7.8	40.6M
Recomputing Enc.	28.1	33.1	28.7	**52.5**	47.9	52.5	**10.4**	**6.6**	52.9M
+ pretrained	**28.4**	**33.6**	**29.1**	**52.5**	**47.4**	**52.1**	10.5	6.7	52.9M

size of the (bidirectional) encoder. The baseline is trained with a learning rate of 10^{-3}. For WMT 2017 German→English with a size of 500, it is trained with the appended newstest sets 2008-2014. For the larger setup and experiments on WMT 2017 English→German, these sets are not included in the training corpus.

For all setups, we measure the number of processed tokens during the training and decoding. Although we report the timings of all experiments on the same type of GPU (a GeForce GTX 1080 Ti), they depend on external factors like parallel running jobs as well. Nevertheless, they do provide some insight into the performance differences of the various setups. We report Bleu, Ter and perplexity (Ppl) scores of an averaged model. It is created by averaging the parameters of the best 4 snapshots of a single training run (see Section 3.5), selected based on Bleu score on the development set. The Ppl is computed on the development set, Bleu and Ter on the development and both test sets.

For some experiments, we train a baseline model until convergence and then switch to alternate setups, ie. the parts of the network that are identical are already trained. We mark these setups as *pretrained*.

6.3 Recalculating the Encoding

In the baseline system, the source sentence is encoded once at the beginning and kept unchanged while generating the target sequence. We evaluate whether it is beneficial to recalculate the encoder states at every decoder time step (see Section 5.1). This modification yields significant improvements (see Table 6.2). The average Bleu score of the pretrained model increases by 2.1 percent points (pps), the average Ter score decreases by 1.3 pp. However, the number of processed source tokens per second both during training and decoding drops by a factor of 8 and 3, respectively (see Table 6.3). Because the required number of iterations until convergence does not decrease, the resulting increase in training time is prohibitively large, preventing further experiments.

Table 6.3: Trainind and decoding speed of recomputing the encoding, measured on WMT 2017 German→English. The size of the encoder/attention/decoder layer is 500.

	Training [tokens/s]	Decoding [tokens/s]	Convergence [iterations]
Baseline	4,870	56	750k
Recomputing Enc.	603	17	800k

6.4 One-Dimensional (1D) Attention

First, we evaluate the general performance of a 1D attention model. In addition, we compare models using gated recurrent units (GRUs) and LSTMs (see Table 6.4).

It is apparent that GRU and LSTM have the same performance. On average, they have the same Bleu and Ter scores. Both setups show a noticeable improvement of 0.9 pp Bleu and 1.0 pp Ter on average over the baseline model. The training of a network with an RNN attention layer is slower than the baseline by a factor of 4, the decoding takes about twice the time (see Table 6.5).

Next, we examine whether initializing the parameters with the decoder state or appending the decoder state to the input yields better results. We try this both for LSTMs and GRUs (see Table 6.4). For LSTM, initializing it with the decoder state outperforms the LSTM attention layer where the decoder state is appended by 0.3 pp Bleu and 0.4 pp Ter on average. However, for the GRU attention layer, both techniques yield the same results. The average Bleu score for initializing the GRU is 0.1 pp above the score for appending it, the average Ter score of both setups is the same. This difference may be explained by the additional memory cell of the LSTM. Because it increases the amount of information that can be stored, the LSTM does not forget part of the decoder state as soon as the GRU. There, appending it to the input can counter the negative effect of the missing memory cell. We expect the benefit of appending the decoder state to become larger for longer sequences.

Finally, we evaluate whether it is best to take the last RNN state or a weighted average (see Section 5.2.1). The results (see Table 6.6) indicate that a weighting mechanism is slightly beneficial overall. This is in contrast to the findings by Zhang et al. [2016]. We assume that this is due to the higher flexibility of our weighting technique. Instead of a simple arithmetic mean, we compute a weighted sum, thus providing the network with more freedom.

Table 6.4: Comparison of GRUs and LSTMs as the attention layer and two ways to pass the decoder state. Trained on WMT 2017 German→English, with an encoder/decoder size of 500. The attention layer of the baseline and the RNNs used in the attention layers have size 1,000. The decoder state is either used to initialize the RNN or appended to its input.

	BLEU [%]			TER [%]			PPL		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	26.9	32.8	27.8	53.6	48.8	52.8	11.7	7.3	41.6M
LSTM (initialized)	28.5	**33.4**	**29.0**	**52.2**	**47.3**	**52.3**	10.1	6.5	48.6M
+ pretrained	28.3	32.9	**29.0**	52.6	47.7	**52.3**	10.2	6.5	48.6M
LSTM (appended)	28.4	32.8	28.4	52.8	48.3	53.0	10.5	6.6	49.6M
+ pretrained	28.3	33.1	28.6	52.3	47.8	52.4	**10.0**	**6.4**	49.6M
GRU (initialized)	28.2	33.1	**29.0**	52.7	47.7	52.4	10.2	6.5	46.1M
+ pretrained	**28.6**	33.2	28.7	52.5	47.8	52.4	10.4	6.6	46.1M
GRU (append)	28.4	33.2	28.6	52.4	47.7	52.6	10.3	6.5	47.1M
+ pretrained	28.2	32.9	**29.0**	52.7	48.0	**52.3**	10.4	6.6	47.1M

Table 6.5: Trainind and decoding speed of a model with a 1D LSTM attention layer, measured on WMT 2017 German→English. The size of the encoder/attention/decoder layer is 1,000.

	Training [tokens/s]	Decoding [tokens/s]	Convergence [iterations]
Baseline	2,796	67	750k
LSTM	680	31	1.000k

6.5 Two-Dimensional (2D) Attention

In the following sections, we will present our results for the setups with a 2DLSTM as the attention mechanism. Here, all models have a bidirectional encoder and a decoder as described in [Bahdanau et al., 2014].

6.5.1 General Performance

The 2DLSTM attention is an extension of the 1D approach. We therefore compare the performance of both setups (see Tables 6.7 and 6.8).

Table 6.6: Comparison of different possibilities to compute the context vector. Trained on IWSLT 2010 German→English with an encoder/decoder size of 500. The baseline has an attention layer of size 500, the RNN used in the attention layer has size 1,000. The GRU in the attention layer is initialized with the decoder state.

	Bleu [%]	Ter [%]	Ppl		# params
	2010	2010	Words	BPE	
Baseline	27.1	51.7	22.6	16.5	27.1M
GRU last	**27.6**	50.7	20.8	15.3	32.6M
GRU weighted	27.3	**50.1**	**19.8**	**14.6**	33.1M

Table 6.7: Comparison of 1D and 2DLSTM as the attention machanism. Trained on WMT 2017 German→English, with an encoder/attention/decoder size of 1,000. The embedding size is 620. The 1D and 2DLSTM used in the attention layer do not have peephole connections. For the 1D attention layer, the decoder state is appended to the input.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	28.3	**33.4**	**28.9**	52.9	**46.9**	**51.5**	9.5	6.2	78.3M
1D-LSTM	**28.6**	33.2	**28.9**	**52.3**	47.8	52.5	9.5	6.2	92.3M
2D-LSTM	28.1	33.0	28.7	52.5	47.7	52.3	**9.3**	**6.1**	119.6M

Table 6.8: Comparison of 1D and 2DLSTM as the attention machanism. Trained on WMT 2017 English→German, with an encoder/attention/decoder size of 1,000. The embedding size is 620. The 1D and 2DLSTM used in the attention layer do not have peephole connections. For the 1D attention layer, the decoder state is appended to the input.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	**25.0**	**28.6**	**23.4**	**57.1**	**52.4**	**59.1**	11.7	5.8	78.3M
1D-LSTM	24.6	27.5	22.8	58.6	54.4	60.3	12.1	6.0	92.3M
2D-LSTM	24.5	28.3	22.7	58.2	53.0	60.0	**11.1**	**5.6**	119.6M

Table 6.9: Trainind and decoding speed of a model with a 2DLSTM attention layer, measured on WMT 2017 German→English. The size of the encoder/attention/decoder layer is 1,000.

	Training [tokens/s]	Decoding [tokens/s]	Convergence [iterations]
Baseline	2,796	67	750k
2DLSTM	799	0.8	820k

Table 6.10: Analysis of the influence of an additional weighting layer on top of a 2DLSTM attention layer. Trained on WMT 2017 German→English, with an encoder/attention/decoder size of 500.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	26.4	31.2	27.1	53.6	48.9	53.3	12.5	7.8	40.6M
2DLSTM	27.3	32.2	27.9	53.3	48.4	53.2	11.1	6.9	57.0M
+ pretrained	**28.0**	**32.8**	**28.2**	**52.2**	47.9	52.5	**10.1**	**6.4**	57.0M
2DLSTM weighted	27.8	32.5	28.1	52.4	**47.5**	**52.4**	10.5	6.6	57.3M
+ pretrained	27.4	32.4	28.0	53.0	48.0	52.6	11.1	7.0	57.3M

On average, a 1D LSTM attention model outperforms a model with a 2DLSTM attention layer by 0.1 pp Bleu. However, the average Ter score of the 2D attention model is better by 0.4 pp. Additionally, because the computation of the 2DLSTM can be performed once at the beginning (see Section 5.3.1) and is highly optimized, the training process is much faster. It takes about 3 times as long as the baseline (see Table 6.9). The decoding is much slower, because there the 2DLSTM has to be recomputed at every time step, as described in Section 5.3.1. Of the 119.6 million parameters, 24 million are used for the lookup table needed to embed the target one-hot vectors before they are appended to the 2DLSTM input. By reusing the matrix used in the decoder, these could be removed.

As for 1D attention, we compare whether it is beneficial to use the last state $\boldsymbol{a}_{J,i}$ as the context vector or to compute a weighted sum. The experiment verifies our finding for 1D attention that a weighted sum yields slightly better results (see Table 6.10). Although the best Bleu scores are reached by taking the last state, the average Bleu across all three test sets and both the pretrained and randomly initialized model is the same as for the architecture where a weighted sum

Table 6.11: Evaluation of the influence of the initialization on 2DLSTM. Trained on WMT 2017 German→English. The baseline system has an encoder/attention/decoder size of 1,000 and an embedding size of 620. The other models have an encoder/decoder and embedding size of 500, the 1D LSTM and 2DLSTM have size 1,000. The pretrained 2DLSTM models are initialized using a converged 1D LSTM attention model. The training corpus of the 1D LSTM and 2DLSTM models contain the newstest sets 2008-2014.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	28.3	33.4	28.9	52.9	46.9	51.5	9.5	6.2	78.3M
1DLSTM, lr=10^{-4}	28.5	33.4	29.0	52.2	47.3	52.3	10.1	6.5	48.6M
2DLSTM, lr=10^{-4}	28.2	32.5	28.6	52.7	48.2	52.8	9.9	6.3	70.5M
+ pretrained	28.5	33.5	29.0	52.4	47.4	51.9	**9.5**	**6.1**	70.5M
2DLSTM, lr=$5 \cdot 10^{-4}$	29.0	33.7	29.5	51.6	47.0	51.5	9.9	6.3	70.5M
+ pretrained	**29.1**	**34.4**	**30.0**	**51.5**	**46.4**	**50.7**	**9.5**	**6.1**	70.5M

is computed. For Ter, the weighting algorithm improves the score by 0.3 pp on average.

Finally, we analyze how the initialization influences the 2DLSTM (see Table 6.11). To this end, we train a 1D LSTM model until convergence and use it to initialize the weights of the 2DLSTM model. The difference to the other experiments where pretrained models are used, is that here, the first dimension of the 2DLSTM is initialized with the trained 1D LSTM weights. The only untrained parameters are therefore those that control the flow of information in the second dimension. During the training, all parameters are updated. In addition to the learning rate of 10^{-4}, we experiment with a learning rate of $5 \cdot 10^{-4}$ as well. Interestingly, the model with the larger learning rate is better than the model with the lower learning rate by 0.9 pp Bleu and 1.1 pp Ter on average. Both pretrained models outperform both the baseline system and the randomly initialized 2DLSTM model. The results indicate that the initialization of the 2DLSTM has a large influence on the final performance of the models. The following section shows that for a random initialization, the 2DLSTM is very sensitive to the choice of learning rate.

6.5.2 Learning Rate

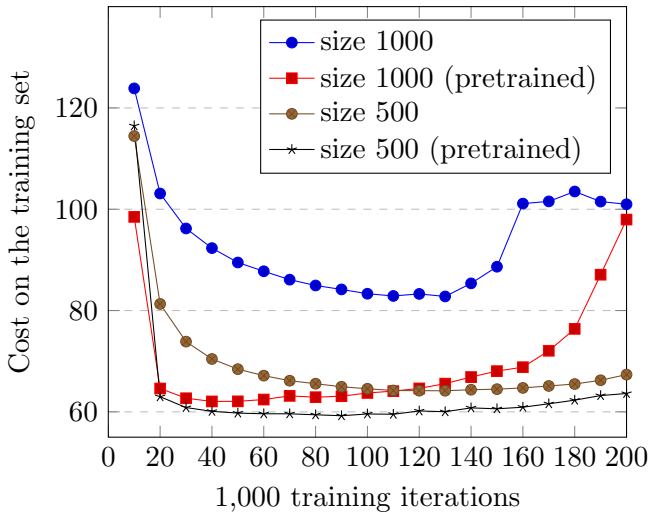
For randomly initialized 2DLSTMs, we evaluate the influence of the learning rate. It becomes apparent that 2D attention is very sensitive to the size of the weight update steps.

In Figure 6.1, one can see both the cost on the training set as well as the perplexity on the development set during the course of a training with a learning rate of 10^{-3}. As depicted there, both start to increase again after a given number of training iterations. This behavior is relatively uncommon for the training of NNs. Usually, over time, the networks tend to overfit on the training corpus, ie. they simply memorize the correct translations and lose their ability to generalize to unseen sentences. This leads to a steadily decreasing cost on the training set and an increase of the perplexity on the development set. The fact that even the cost on the training set increases is an indicator of a problem during training. If the function space is very uneven and has many dips and peeks, an update to the parameter values that is too large might cause the network to jump over a local optimum. Even though training procedures such as Adam try to avoid this problem (see Section 3.4.2), an unfavorable function space may still cause the network to climb up hills by constantly overshooting local optima. This situation is described in more detail in Section 3.4.2 and the corresponding Figure 3.8.

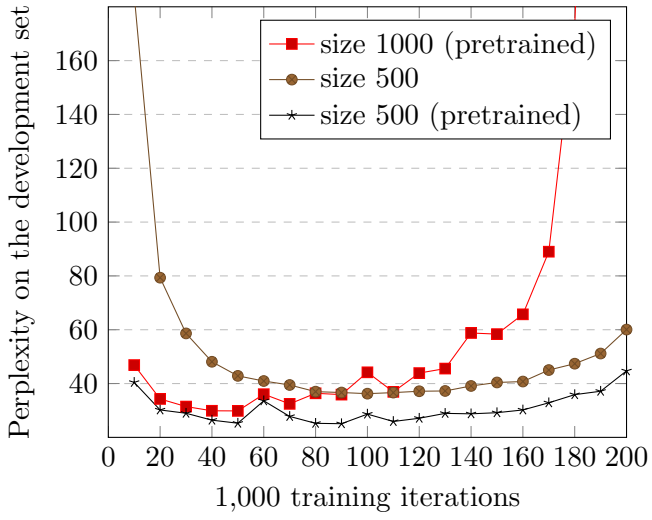
The behavior depicted in Figure 6.1 has been recorded based on an initialization using a gaussian normal distribution with a standard deviation of 10^{-2}. However, a xavier initialization (see Section 3.4.2) yields similar results. Additionally, training a 2D attention layer with a learning rate of 10^{-4} for 570,000 iterations and only then switching to the higher learning rate of 10^{-3} does not help either. As soon as the learning rate is increased, the performance of the model decreases.

This indicates that the gradients with respect to the parameters within a 2D attention layer are very noisy. We have verified this assessment by modifying the batch size. While a 2D attention model with a learning rate of $5 \cdot 10^{-4}$ trains correctly with a batch size of 50, it fails to learn anything with a batch size of 25. For the larger batch size, the noise of the gradients mostly cancels out because it is averaged across 50 training samples. By reducing the batch size, the noise has more influence and prohibits a successful training.

We therefore train our models using a learning rate of 10^{-4} and a batch size of 50, as defined in Section 6.2. This avoids the aforementioned issues, but increases the necessary number of training iterations until convergence and therefore the complete training time. The positive effect of initializing the 2DLSTM with a trained 1D LSTM attention model described in the previous section was only discovered at the end of the thesis, so it could not be used in the other experiments.



(a) Cost on the training set



(b) Perplexity on the development set. The Ppl of the randomly initialized model with size 1,000 never drops below 210.

Figure 6.1: Training a 2DLSTM with learning rate 10^{-3} leads to an increasing training cost and perplexity. We experimented both with 2DLSTM cells of size 500 and 1,000. Trained and evaluated based on corpus WMT 2017 English→German.

Table 6.12: Comparison of 2DLSTM sizes. Trained on WMT 2017 English→German, with an encoder/decoder size of 1,000. The baseline has an attention layer of size 1,000, the size of the 2DLSTM is specified individually.

	Bleu [%]			Ter [%]			Ppl		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline	**25.0**	**28.6**	**23.4**	**57.1**	**52.4**	**59.1**	11.7	5.8	78.3M
2DLSTM size 1,000	24.5	28.3	22.7	58.2	53.0	60.1	**11.1**	**5.6**	119.6M
2DLSTM size 500	23.9	27.1	22.1	58.6	54.2	60.7	12.9	6.3	105.8M

Table 6.13: Trainind and decoding speed of a model with a 2DLSTM attention layer 500, measured on WMT 2017 English→German. The size of the encoder/decoder layer is 1,000. The first two experiments have an attention layer of size 1,000. In the third experiment, it is reduced to size 500.

	Training [tokens/s]	Decoding [tokens/s]	Convergence [iterations]
Baseline	2,987	64	600k
2DLSTM	788	0.8	870k
2DLSTM size 500	1,243	2.4	1.200k

6.5.3 Model Size

We want to analyze whether the 2DLSTM attention mechanism is used to its full potential. To this end, we perform an experiment with a reduced 2DLSTM size. The drop in Bleu and increase in Ter is 0.8 pp and 0.7 pp, respectively. This is an indicator that the 2DLSTM is in fact using all or most of the available parameters. It should be noted, that even the smaller 2DLSTM attention model outperforms the baseline system. By reducing the 2DLSTM size, the training speed is increased to half the speed of the baseline. However, the required number of iterations until convergence increases. The decoding speed is increased as well, even though it is still slowed down by the repeated recalculation of the 2DLSTM (see Table 6.13).

Table 6.14: Evaluation of the performance of a 2D seq2seq model. Trained on WMT 2017 German→English. The two baseline systems have an encoder/attention/decoder layer with size 500 and 1,000. The size of the 2D seq2seq model is specified individually. The experiment marked with *weighted* has an additional 2D encoder and a weighting mechanism on top of the 2DLSTM.

	BLEU [%]			TER [%]			PPL		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
Baseline size 500	26.4	31.2	27.1	53.6	48.9	53.3	12.5	7.8	40.6M
2D seq2seq size 500	25.9	29.4	26.0	55.5	51.4	55.7	12.3	7.7	41.2M
+ weighted	26.9	31.3	26.3	54.0	49.2	56.6	11.2	7.1	46.4M
Baseline size 1,000[2]	**28.3**	**33.4**	**28.9**	**52.9**	**46.9**	**51.5**	**9.5**	**6.2**	78.3M
2D seq2seq size 1,000	27.1	31.4	27.4	53.9	49.4	54.3	10.3	6.6	91.8M

Table 6.15: Trainind and decoding speed of a 2D seq2seq model, measured on WMT 2017 German→English. The size of the encoder/attention/decoder layer in the baseline system is 1,000. The 2DLSTM has size 1,000 as well

	Training [tokens/s]	Decoding [tokens/s]	Convergence [iterations]
Baseline	2,796	67	750k
2D seq2seq	1,023	0.8	1.200k

6.6 Two-Dimensional Sequence to Sequence (2D seq2seq)

The previous modifications only replace the encoding or attention layer. In this section, we will examine the performance of the 2D seq2seq model that only consists of a single 2DLSTM (see Section 5.4).

We first compare the baseline model with two 2D seq2seq setups that have either 500 or 1,000 nodes in the 2DLSTM layer (see Table 6.14). Both 2D seq2seq models are outperformed by the corresponding baselines by 1.1 and 1.6 pp BLEU, and 2.3 and 2.1 pp TER, respectively. Nevertheless, the performance proves that, as we hypothesized, the 2DLSTM is able of internalizing the encoder, attention mechanism and decoder. Compared with the baseline, the average training time increases by a factor of 3 (see Table 6.15). During the decoding, the optimization described in Section 5.3.1 is not possible, resulting in the slow speed.

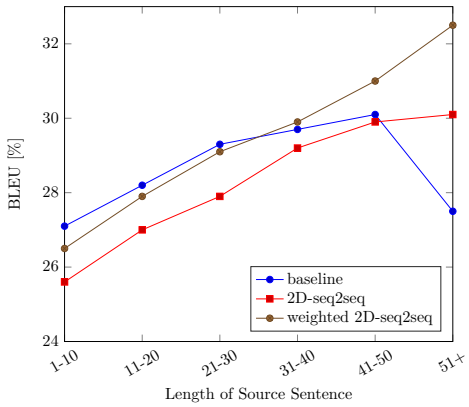
Additionally, it should be noted that the 2D seq2seq network is at a significant disadvantage. Because we remove the bidirectional encoder and the 2DLSTM is unidirectional, the network has no knowledge of the words f_{j+1}^{J} while processing word f_j.

We try to make the 2D seq2seq model as strong as possible by equipping it with both a 2D encoder (see Section 5.4.1) and a weighting algorithm (see Section 5.4.2). Using this modification, the small model with size 500 yields results that are competitive compared to the baseline (see Table 6.14). The average BLEU score of the baseline is better by 0.1 pp, the average TER is better by 1.3 pp. In the previous sections, we have shown that the improvement gained by a weighting algorithm is relatively small for 1D and 2DLSTM attention. This indicates that the improvement is caused by the additional 2D encoder. Because the source sentence is inverted for one of the two dimensions, the used source representations encode the whole source sentence with focus on the corresponding source words. This is consistent with a preliminary experiment where we use a 1D bidirectional encoder and no weighting algorithm. There, the average performance compared with a 2D seq2seq model with no encoder improved by 1 pp on both BLEU and TER.

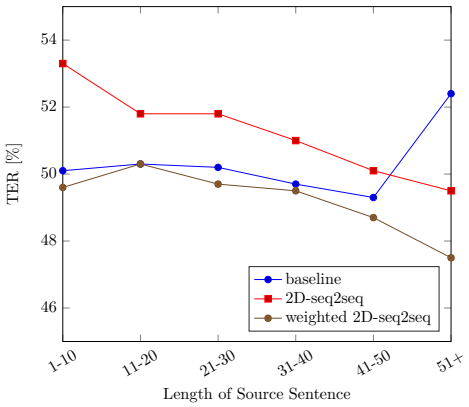
The 2D seq2seq model internally stores the whole target history and can modify the source representation based on the current decoding step. To evaluate whether this helps the model to avoid over- and under-translations, we score the hypotheses with respect to the source length. To make the scores more reliable, we concatenate the three sets newstest2015, newstest2016 and newstest2017 for German→English translation. We separate them based on the number of words in the source sentence (1-10, 11-20, 21-30, 31-40, 41-50 and 51 or more words), yielding groups of 1455, 3081, 2133, 990, 344 and 169 sentences, respectively. As shown in Figure 6.2, the performance of the baseline system drops for sequences longer than 50 words. This can be explained with the length of the training sequences. Because only sequences shorter than 50 subword units have been used, the model has never seen longer sentences. However, both the simple 2D seq2seq model and the architecture with an additional encoder and weighting algorithm do not suffer from long sequences. This indicates that the model is able to store some form of fertility information internally, helping it to avoid over- and under-translations.

Finally, we evaluate the performance of a bidirectional 2D seq2seq model (see Table 6.16). On average, the BLEU score of both models is the same. The average TER score of the bidirectional 2D seq2seq model is better by 0.2 pp. That no significant improvement is gained by adding the second direction agrees with the results for 1D LSTM attention.

[2]This model has an encoder size of 620.



(a) Development of the BLEU score w.r.t. the sequence length



(b) Development of the TER score w.r.t. the sequence length

Figure 6.2: Performance of the baseline and two 2D seq2seq models w.r.t the source sequence length on the concatenation of the three sets newstest2015, newstest2016 and newstest2017 for German→English translation.

Table 6.16: Comparison of a simple and an extended 2D seq2seq model. Trained on WMT 2017 German→English, with an encoder/attention/decoder size of 500.

	BLEU [%]			TER [%]			PPL		# params
	2015	2016	2017	2015	2016	2017	Words	BPE	
2D seq2seq size 500	**25.9**	29.4	**26.0**	**55.5**	51.4	55.7	**12.3**	**7.6**	41.2M
+ bidirectional	25.6	**29.6**	**26.0**	**55.5**	**51.1**	**55.4**	12.5	**7.6**	44.9M

Chapter 7

Conclusion and Outlook

In this bachelor thesis, several new network topologies have been proposed and tested. We now want to summarize our findings and propose further avenues of work.

7.1 Conclusion

We have shown that the ability to recalculate the encoder states based on the decoder state yields significant improvements over a classical encoder-decoder architecture with attention. However, this modification increases the required training time per iteration by a factor of 9. Therefore, this technique is not reasonably usable in practice.

We have been able to verify that using one-dimensional (1D) recurrent neural networks (RNNs) as the attention layer is beneficial, as stated by Zhang et al. [2016]. However, using a weighting algorithm that can compute any weighted sum yields further improvements, even though the authors report negative results for a simple arithmetic mean. Because 1D attention is slow to train as well, the practical application is limited.

We have shown that two-dimensional LSTM (2DLSTM) is a powerful tool. By employing it in the attention layer, we are able to speed up the training, because the whole 2DLSTM can be computed once and highly parallelized. In the direct comparison, two-dimensional (2D) attention performs similar to an attention layer with only one dimension. We have deduced that the gradients of 2DLSTM attention models with respect to the parameters within the attention layer are very noisy. This hinders the training, because it requires the learning rate to be low. Gaining a speedup by reducing the 2DLSTM size is not advisable as it results in a significant loss in translation quality.

Finally, we have proposed a 2D sequence to sequence (2D seq2seq) model as a novel architecture. By unifying the encoder, attention layer and decoder within a single 2DLSTM, we are able to reach a training speed of more than one third of the baseline. The resulting model are outperformed by the baseline by about 1.3 percent point (pp) Bleu and 2.2 pp Ter on average. By adding an additional 2D

encoder and a weighting algorithm, the network becomes stronger, the BLEU score of the weighted 2D seq2seq model is as good as the score of the baseline. For TER, the baseline still outperforms the improved model by 1.3 pp. These results are especially noteworthy as the 2D seq2seq architecture has not yet been significantly fine-tuned.

By evaluating the models for different source lengths, we have shown that the proposed models do in fact reduce the risk of over- and under-translations. This indicates that the network does learn the concept of fertility.

7.2 Outlook

For 1D RNNs as the attention mechanism, we find that initializing them with the decoder state and appending it to their input yields comparable results. We expect the performance of a solely initialized model to decrease as the sentence length increases. We therefore propose to further evaluate how the information stored in the decoder state can be best passed on to the attention RNN. Furthermore, we would like to analyze why using a bidirectional RNN hinders the training process.

For 2DLSTM as the attention layer, we propose to investigate why the gradient is as noisy as it is. Once a possibility to reduce this noise is found, the learning rate could be increased, resulting in a faster convergence. This would allow it to further fine-tune the hyperparameters such as the layer size or the specific setup of the weighting algorithm. By experimenting with different initialization and training schemes, it might be possible to further increase the performance of 2DLSTM, making them a reasonable replacement for the usual attention method. To this end, it may be necessary to speed up the computations of RNNs in general.

We find the experiments with only one or two 2DLSTM layers as the single element of the network especially promising. The fact that the performance difference between the 2D seq2seq model and the baseline was no more than 1.6 pp BLEU and 2.3 pp TER, even though it has to internalize the encoder, attention layer and the decoder, is quite fascinating. A more intense study of 2DLSTMs will probably reveal possibilities to further improve their performance. A specific aspect we would like to highlight for further research is the option to add peephole connections to the 2DLSTM.

Appendix A

Appendix

A.1 Derivations

To determine the direction of the weight updates, the gradient of the loss function has to be computed with respect to every parameter. In the following sections we first show a general derivation for matrix-vector products. Then, we provide the derivations for long short-term memory (LSTM) and 2DLSTM. To simplify the notation, we perform the derivations always based on scalars. In some cases, this, as well as the time recurrence for LSTM and 2DLSTM, leads to multiple gradients for the same scalar. These have to be added prior to following computations.

A.1.1 Matrix-Vector Products

We write $\boldsymbol{W}_{m,n}$ to denote the scalar in $\boldsymbol{W}$ in row m and column n. Similarly, $\boldsymbol{x}_n$ is the scalar in row n of $\boldsymbol{x}$. Using

$$(\boldsymbol{W}\boldsymbol{x})_m = \sum_n \boldsymbol{W}_{m,n}\boldsymbol{x}_n \tag{A.1}$$

one can compute the gradients as follows:

$$\frac{\partial(\boldsymbol{W}\boldsymbol{x})_m}{\partial \boldsymbol{W}_{m,n}} = \frac{\partial \sum_{n'} \boldsymbol{W}_{m,n'}\boldsymbol{x}_{n'}}{\partial \boldsymbol{W}_{m,n}} \tag{A.2}$$

$$= \frac{\partial(\boldsymbol{W}_{m,n}\boldsymbol{x}_n)}{\partial \boldsymbol{W}_{m,n}} \tag{A.3}$$

$$= \boldsymbol{x}_n \tag{A.4}$$

$$\frac{\partial(\boldsymbol{W}\boldsymbol{x})_m}{\partial \boldsymbol{x}_n} = \frac{\partial \sum_{n'} \boldsymbol{W}_{m,n'}\boldsymbol{x}_{n'}}{\partial \boldsymbol{x}_n} \tag{A.5}$$

$$= \frac{\partial(\boldsymbol{W}_{m,n}\boldsymbol{x}_n)}{\partial \boldsymbol{x}_n} \tag{A.6}$$

$$= \boldsymbol{W}_{m,n} \tag{A.7}$$

A.1.2 Long Short-Term Memory (LSTM)

The formulas of an LSTM can easily be differentiated by using the chain rule. To increase the readability, we use the simplified notation introduced at the end of Section 3.3.1. Therefore, we have to differentiate the Equations 3.13 to 3.18:

$$\boldsymbol{f}_t = \sigma(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T) \tag{A.8}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T) \tag{A.9}$$

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T) \tag{A.10}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \tilde{\boldsymbol{c}}_t \tag{A.11}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T) \tag{A.12}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t) \tag{A.13}$$

We write $\boldsymbol{x}_n$ to denote the scalar in vector $\boldsymbol{x}$ in row n. E is the value of the loss function $\mathcal{L}$. Once the derivation of the LSTM cell is needed, all layers that are above it have already been differentiated. $\frac{\partial E}{\partial \boldsymbol{h}_{tn}}$ is therefore given. One can now compute the derivations in reverse order. For Equation A.13:

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{o}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \frac{\partial \boldsymbol{h}_{tn}}{\partial \boldsymbol{o}_{tn}} \\
&\quad \overset{\text{Eq. A.13}}{=} \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \frac{\partial(\boldsymbol{o}_{tn} \cdot \tanh(\boldsymbol{c}_{tn}))}{\partial \boldsymbol{o}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \tanh(\boldsymbol{c}_{tn})
\end{aligned} \tag{A.14}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{c}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \frac{\partial \boldsymbol{h}_{tn}}{\partial \boldsymbol{c}_{tn}} \\
&\quad \overset{\text{Eq. A.13}}{=} \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \frac{\partial(\boldsymbol{o}_{tn} \cdot \tanh(\boldsymbol{c}_{tn}))}{\partial \boldsymbol{c}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{tn}} \cdot \boldsymbol{o}_{tn} \cdot (1 - \tanh^2(\boldsymbol{c}_{tn}))
\end{aligned} \tag{A.15}$$

For Equation A.12:

$$\begin{aligned}
&\frac{\partial E}{\partial (\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{o}_{tn}} \cdot \frac{\partial \boldsymbol{o}_{tn}}{\partial (\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n} \\
&\quad \overset{\text{Eq. A.12}}{=} \frac{\partial E}{\partial \boldsymbol{o}_{tn}} \cdot \frac{\partial \sigma\big((\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n\big)}{\partial (\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{o}_{tn}} \cdot \sigma\big((\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n\big) \cdot \big[1 - \sigma\big((\boldsymbol{M}_o[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_t^T]^T)_n\big)\big] \\
&\quad \overset{\text{Eq. A.12}}{=} \frac{\partial E}{\partial \boldsymbol{o}_{tn}} \cdot \boldsymbol{o}_{tn} \cdot (1 - \boldsymbol{o}_{tn})
\end{aligned} \tag{A.16}$$

For Equation A.11:

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{f}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial \boldsymbol{c}_{tn}}{\partial \boldsymbol{f}_{tn}} \\
&\quad \overset{\text{Eq. A.11}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial (\boldsymbol{f}_{tn} \cdot \boldsymbol{c}_{t-1n} + \boldsymbol{i}_{tn} \cdot \tilde{\boldsymbol{c}}_{tn})}{\partial \boldsymbol{f}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \boldsymbol{c}_{t-1n}
\end{aligned} \tag{A.17}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{c}_{t-1n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial \boldsymbol{c}_{tn}}{\partial \boldsymbol{c}_{t-1n}} \\
&\quad \overset{\text{Eq. A.11}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial (\boldsymbol{f}_{tn} \cdot \boldsymbol{c}_{t-1n} + \boldsymbol{i}_{tn} \cdot \tilde{\boldsymbol{c}}_{tn})}{\partial \boldsymbol{c}_{t-1n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \boldsymbol{f}_{tn}
\end{aligned} \tag{A.18}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{i}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial \boldsymbol{c}_{tn}}{\partial \boldsymbol{i}_{tn}} \\
&\quad \overset{\text{Eq. A.11}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial (\boldsymbol{f}_{tn} \cdot \boldsymbol{c}_{t-1n} + \boldsymbol{i}_{tn} \cdot \tilde{\boldsymbol{c}}_{tn})}{\partial \boldsymbol{i}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \tilde{\boldsymbol{c}}_{tn}
\end{aligned} \tag{A.19}$$

$$
\begin{aligned}
&\frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial \boldsymbol{c}_{tn}}{\partial \tilde{\boldsymbol{c}}_{tn}} \\
&\quad \overset{\text{Eq. A.11}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \frac{\partial (\boldsymbol{f}_{tn} \cdot \boldsymbol{c}_{t-1n} + \boldsymbol{i}_{tn} \cdot \tilde{\boldsymbol{c}}_{tn})}{\partial \tilde{\boldsymbol{c}}_{tn}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{tn}} \cdot \boldsymbol{i}_{tn}
\end{aligned}
\tag{A.20}
$$

For Equation A.10:

$$
\begin{aligned}
&\frac{\partial E}{\partial (\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T)_n} \\
&\quad = \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{tn}} \cdot \frac{\partial \tilde{\boldsymbol{c}}_{tn}}{\partial (\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T)_n} \\
&\quad \overset{\text{Eq. A.10}}{=} \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{tn}} \cdot \frac{\partial \tanh\left((\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T)_n\right)}{\partial (\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T)_n} \\
&\quad = \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{tn}} \cdot \left[1 - \tanh^2\left((\boldsymbol{M}_c[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T]^T)_n\right)\right] \\
&\quad \overset{\text{Eq. A.10}}{=} \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{tn}} \cdot (1 - \tilde{\boldsymbol{c}}_{tn}^2)
\end{aligned}
\tag{A.21}
$$

For Equation A.9:

$$
\begin{aligned}
&\frac{\partial E}{\partial \left(\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\right)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{i}_{tn}} \cdot \frac{\partial \boldsymbol{i}_{tn}}{\partial \left(\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\right)_n} \\
&\quad \overset{\text{Eq. A.9}}{=} \frac{\partial E}{\partial \boldsymbol{i}_{tn}} \cdot \frac{\partial \sigma\left((\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n\right)}{\partial \left(\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\right)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{i}_{tn}} \cdot \sigma\left((\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n\right) \cdot \left[1 - \sigma\left((\boldsymbol{M}_i[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n\right)\right] \\
&\quad \overset{\text{Eq. A.9}}{=} \frac{\partial E}{\partial \boldsymbol{i}_{tn}} \cdot \boldsymbol{i}_{tn} \cdot (1 - \boldsymbol{i}_{tn})
\end{aligned}
\tag{A.22}
$$

For Equation A.8:

$$
\begin{aligned}
&\frac{\partial E}{\partial \big(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{f}_{tn}} \cdot \frac{\partial \boldsymbol{f}_{tn}}{\partial \big(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\big)_n} \\
&\quad \overset{\text{Eq. A.8}}{=} \frac{\partial E}{\partial \boldsymbol{f}_{tn}} \cdot \frac{\partial \sigma\big((\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n\big)}{\partial \big(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T\big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{f}_{tn}} \cdot \sigma(\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n \cdot \big[1 - \sigma\big((\boldsymbol{M}_f[\boldsymbol{x}_t^T, \boldsymbol{h}_{t-1}^T, \boldsymbol{c}_{t-1}^T]^T)_n\big)\big] \\
&\quad \overset{\text{Eq. A.8}}{=} \frac{\partial E}{\partial \boldsymbol{f}_{tn}} \cdot \boldsymbol{f}_{tn} \cdot (1 - \boldsymbol{f}_{tn})
\end{aligned}
\tag{A.23}
$$

Afterwards, the derivation of the loss function with respect to the weight matrices and $\boldsymbol{x}_t$, $\boldsymbol{h}_{t-1}$ and $\boldsymbol{c}_{t-1}$ can be determined as described in Appendix A.1.1. It should be noted that the aforementioned addition of multiple gradients for the same scalar applies to the gradient of $\boldsymbol{c}_t$ as well. The gradient in Equation A.15 has to be added to the one computed in Equation A.18 during the earlier backpropagation step.

A.1.3 Two-Dimensional LSTM (2DLSTM)

The derivation of the 2DLSTM is similar to the derivation of an LSTM cell with only one dimension (see Appendix A.1.2). We again determine the derivations based on the simplified notation defined at the end of Section 3.3.2. Therefore, we have to differentiate the Equations 3.26 to 3.32:

$$\boldsymbol{f}_{t,t'} = \sigma(\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T) \tag{A.24}$$

$$\boldsymbol{i}_{t,t'} = \sigma(\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T) \tag{A.25}$$

$$\boldsymbol{\lambda}_{t,t'} = \sigma(\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T) \tag{A.26}$$

$$\tilde{\boldsymbol{c}}_{t,t'} = \tanh\left(\boldsymbol{M}_c[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T]^T\right) \tag{A.27}$$

$$\boldsymbol{c}_{t,t'} = \boldsymbol{f}_{t,t'} \circ (\boldsymbol{\lambda}_{t,t'} \circ \boldsymbol{c}_{t-1,t'} + (\boldsymbol{1} - \boldsymbol{\lambda}_{t,t'}) \circ \boldsymbol{c}_{t,t'-1}) + \boldsymbol{i}_{t,t'} \circ \tilde{\boldsymbol{c}}_{t,t'} \tag{A.28}$$

$$\boldsymbol{o}_{t,t'} = \sigma(\boldsymbol{M}_o[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t,t'}^T]^T) \tag{A.29}$$

$$\boldsymbol{h}_{t,t'} = \boldsymbol{o}_{t,t'} \circ \tanh\left(\boldsymbol{c}_{t,t'}\right) \tag{A.30}$$

We write $\boldsymbol{x}_n$ to denote the scalar in vector $\boldsymbol{x}$ in row n. E is the value of the loss function $\mathcal{L}$. Once the derivation of the LSTM is needed, all layers above it have already been differentiated. $\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{h}_{t,t'_n}}$ is therefore given. One can now compute the derivations in reverse order. For Equation A.30:

$$
\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{o}_{t,t'\,n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \frac{\partial \boldsymbol{h}_{t,t'\,n}}{\partial \boldsymbol{o}_{t,t'\,n}} \\
&\quad \overset{\text{Eq. A.30}}{=} \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \frac{\partial (\boldsymbol{o}_{t,t'\,n} \cdot \tanh(\boldsymbol{c}_{t,t'\,n}))}{\partial \boldsymbol{o}_{t,t'\,n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \tanh(\boldsymbol{c}_{t,t'\,n})
\end{aligned}
\tag{A.31}
$$

$$
\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \frac{\partial \boldsymbol{h}_{t,t'\,n}}{\partial \boldsymbol{c}_{t,t'\,n}} \\
&\quad \overset{\text{Eq. A.30}}{=} \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \frac{\partial (\boldsymbol{o}_{t,t'\,n} \cdot \tanh(\boldsymbol{c}_{t,t'\,n}))}{\partial \boldsymbol{c}_{t,t'\,n}} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{h}_{t,t'\,n}} \cdot \boldsymbol{o}_{t,t'\,n} \cdot (1 - \tanh^2(\boldsymbol{c}_{t,t'\,n}))
\end{aligned}
\tag{A.32}
$$

For Equation A.29:

$$
\begin{aligned}
&\frac{\partial E}{\partial \Big(\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{o}_{t,t'\,n}} \cdot \frac{\partial \boldsymbol{o}_{t,t'\,n}}{\partial \Big(\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T\Big)_n} \\
&\quad \overset{\text{Eq. A.29}}{=} \frac{\partial E}{\partial \boldsymbol{o}_{t,t'\,n}} \cdot \frac{\partial \sigma\Big((\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T)_n\Big)}{\partial \Big(\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{o}_{t,t'\,n}} \cdot \sigma\big((\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T)_n\big) \\
&\qquad \cdot \big[1 - \sigma\big((\boldsymbol{M}_o[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}, \boldsymbol{c}^T_{t,t'}]^T)_n\big)\big] \\
&\quad \overset{\text{Eq. A.29}}{=} \frac{\partial E}{\partial \boldsymbol{o}_{t,t'\,n}} \cdot \boldsymbol{o}_{t,t'\,n} \cdot (1 - \boldsymbol{o}_{t,t'\,n})
\end{aligned}
\tag{A.33}
$$

For Equation A.28:

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{f}_{t,t'_n}} \qquad (A.34)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'_n}}{\partial \boldsymbol{f}_{t,t'_n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \big(\boldsymbol{f}_{t,t'_n} \cdot (\boldsymbol{\lambda}_{t,t'_n} \cdot \boldsymbol{c}_{t-1,t'_n} + (1-\boldsymbol{\lambda}_{t,t'_n}) \cdot \boldsymbol{c}_{t,t'-1_n}) + \boldsymbol{i}_{t,t'_n} \cdot \tilde{\boldsymbol{c}}_{t,t'_n}\big)}{\partial \boldsymbol{f}_{t,t'_n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \big(\boldsymbol{\lambda}_{t,t'_n} \cdot \boldsymbol{c}_{t-1,t'_n} + (1-\boldsymbol{\lambda}_{t,t'-1_n}) \cdot \boldsymbol{c}_{t,t'-1_n}\big)
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{c}_{t-1,t'_n}} \qquad (A.35)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'_n}}{\partial \boldsymbol{c}_{t-1,t'_n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \big(\boldsymbol{f}_{t,t'_n} \cdot (\boldsymbol{\lambda}_{t,t'_n} \cdot \boldsymbol{c}_{t-1,t'_n} + (1-\boldsymbol{\lambda}_{t,t'_n}) \cdot \boldsymbol{c}_{t,t'-1_n}) + \boldsymbol{i}_{t,t'_n} \cdot \tilde{\boldsymbol{c}}_{t,t'_n}\big)}{\partial \boldsymbol{c}_{t-1,t'_n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \boldsymbol{f}_{t,t'_n} \cdot \boldsymbol{\lambda}_{t,t'_n}
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{c}_{t,t'-1_n}} \qquad (A.36)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'_n}}{\partial \boldsymbol{c}_{t,t'-1_n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \big(\boldsymbol{f}_{t,t'_n} \cdot (\boldsymbol{\lambda}_{t,t'_n} \cdot \boldsymbol{c}_{t-1,t'_n} + (1-\boldsymbol{\lambda}_{t,t'_n}) \cdot \boldsymbol{c}_{t,t'-1_n}) + \boldsymbol{i}_{t,t'_n} \cdot \tilde{\boldsymbol{c}}_{t,t'_n}\big)}{\partial \boldsymbol{c}_{t,t'-1_n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \boldsymbol{f}_{t,t'_n} \cdot (1-\boldsymbol{\lambda}_{t,t'_n})
\end{aligned}$$

$$\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{\lambda}_{t,t'_n}} \qquad (A.37)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'_n}}{\partial \boldsymbol{\lambda}_{t,t'_n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \frac{\partial \big(\boldsymbol{f}_{t,t'_n} \cdot (\boldsymbol{\lambda}_{t,t'_n} \cdot \boldsymbol{c}_{t-1,t'_n} + (1-\boldsymbol{\lambda}_{t,t'_n}) \cdot \boldsymbol{c}_{t,t'-1_n}) + \boldsymbol{i}_{t,t'_n} \cdot \tilde{\boldsymbol{c}}_{t,t'_n}\big)}{\partial \boldsymbol{\lambda}_{t,t'_n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'_n}} \cdot \boldsymbol{f}_{t,t'_n} \cdot \big(\boldsymbol{c}_{t-1,t'_n} - \boldsymbol{c}_{t,t'-1_n}\big)
\end{aligned}$$

$$
\begin{aligned}
&\frac{\partial E}{\partial \boldsymbol{i}_{t,t'\,n}} \qquad (A.38)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'\,n}}{\partial \boldsymbol{i}_{t,t'\,n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \frac{\partial \Big(\boldsymbol{f}_{t,t'\,n} \cdot (\boldsymbol{\lambda}_{t,t'\,n} \cdot \boldsymbol{c}_{t-1,t'\,n} + (1 - \boldsymbol{\lambda}_{t,t'\,n}) \cdot \boldsymbol{c}_{t,t'-1\,n}) + \boldsymbol{i}_{t,t'\,n} \cdot \tilde{\boldsymbol{c}}_{t,t'\,n}\Big)}{\partial \boldsymbol{i}_{t,t'\,n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \tilde{\boldsymbol{c}}_{t,t'\,n}
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}} \qquad (A.39)\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \frac{\partial \boldsymbol{c}_{t,t'\,n}}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}}\\
&\quad \overset{\text{Eq. A.28}}{=} \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \frac{\partial \Big(\boldsymbol{f}_{t,t'\,n} \cdot (\boldsymbol{\lambda}_{t,t'\,n} \cdot \boldsymbol{c}_{t-1,t'\,n} + (1 - \boldsymbol{\lambda}_{t,t'\,n}) \cdot \boldsymbol{c}_{t,t'-1\,n}) + \boldsymbol{i}_{t,t'\,n} \cdot \tilde{\boldsymbol{c}}_{t,t'\,n}\Big)}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}}\\
&\quad = \frac{\partial E}{\partial \boldsymbol{c}_{t,t'\,n}} \cdot \boldsymbol{i}_{t,t'\,n}
\end{aligned}
$$

For Equation A.27:

$$
\begin{aligned}
&\frac{\partial E}{\partial \Big(\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T\Big)_n} \qquad (A.40)\\
&\quad = \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}} \cdot \frac{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}}{\partial \Big(\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T\Big)_n}\\
&\quad \overset{\text{Eq. A.27}}{=} \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}} \cdot \frac{\partial \tanh\Big((\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T)_n\Big)}{\partial \Big(\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T\Big)_n}\\
&\quad = \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}} \cdot \left[1 - \tanh^2\left((\boldsymbol{M}_c[\boldsymbol{x}^T_{t,t'}, \boldsymbol{h}^T_{t-1,t'}, \boldsymbol{h}^T_{t,t'-1}]^T)_n\right)\right]\\
&\quad \overset{\text{Eq. A.27}}{=} \frac{\partial E}{\partial \tilde{\boldsymbol{c}}_{t,t'\,n}} \cdot (1 - \tilde{\boldsymbol{c}}^2_{t,t'\,n})
\end{aligned}
$$

For Equation A.26:

$$
\begin{aligned}
&\frac{\partial E}{\partial\Big(\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{\lambda}_{t,t'}{}_n} \cdot \frac{\partial \boldsymbol{\lambda}_{t,t'}{}_n}{\partial\Big(\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad \overset{\text{Eq. A.26}}{=} \frac{\partial E}{\partial \boldsymbol{\lambda}_{t,t'}{}_n} \cdot \frac{\partial \sigma\Big((\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\Big)}{\partial\Big(\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{\lambda}_{t,t'}{}_n} \cdot \sigma\big((\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big) \\
&\qquad \cdot \big[1 - \sigma\big((\boldsymbol{M}_\lambda[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big)\big] \\
&\quad \overset{\text{Eq. A.26}}{=} \frac{\partial E}{\partial \boldsymbol{\lambda}_{t,t'}{}_n} \cdot \boldsymbol{\lambda}_{t,t'}{}_n \cdot (1 - \boldsymbol{\lambda}_{t,t'}{}_n)
\end{aligned}
\tag{A.41}
$$

For Equation A.25:

$$
\begin{aligned}
&\frac{\partial E}{\partial\Big(\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{i}_{t,t'}{}_n} \cdot \frac{\partial \boldsymbol{i}_{t,t'}{}_n}{\partial\Big(\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad \overset{\text{Eq. A.25}}{=} \frac{\partial E}{\partial \boldsymbol{i}_{t,t'}{}_n} \cdot \frac{\partial \sigma\Big((\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\Big)}{\partial\Big(\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\quad = \frac{\partial E}{\partial \boldsymbol{i}_{t,t'}{}_n} \cdot \sigma\big((\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big) \\
&\qquad \cdot \big[1 - \sigma\big((\boldsymbol{M}_i[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big)\big] \\
&\quad \overset{\text{Eq. A.25}}{=} \frac{\partial E}{\partial \boldsymbol{i}_{t,t'}{}_n} \cdot \boldsymbol{i}_{t,t'}{}_n \cdot (1 - \boldsymbol{i}_{t,t'}{}_n)
\end{aligned}
\tag{A.42}
$$

For Equation A.24:

$$\begin{aligned}
&\frac{\partial E}{\partial\Big(\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&= \frac{\partial E}{\partial \boldsymbol{f}_{t,t'n}} \cdot \frac{\partial \boldsymbol{f}_{t,t'n}}{\partial\Big(\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&\overset{\text{Eq. A.24}}{=} \frac{\partial E}{\partial \boldsymbol{f}_{t,t'n}} \cdot \frac{\partial \sigma\Big((\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\Big)}{\partial\Big(\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T\Big)_n} \\
&= \frac{\partial E}{\partial \boldsymbol{f}_{t,t'n}} \cdot \sigma\big((\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big) \\
&\quad \cdot \big[1 - \sigma\big((\boldsymbol{M}_f[\boldsymbol{x}_{t,t'}^T, \boldsymbol{h}_{t-1,t'}^T, \boldsymbol{h}_{t,t'-1}^T, \boldsymbol{c}_{t-1,t'}^T, \boldsymbol{c}_{t,t'-1}^T]^T)_n\big)\big] \\
&\overset{\text{Eq. A.24}}{=} \frac{\partial E}{\partial \boldsymbol{f}_{t,t'n}} \cdot \boldsymbol{f}_{t,t'n} \cdot (1 - \boldsymbol{f}_{t,t'n})
\end{aligned} \tag{A.43}$$

One can compute the derivations for the weight matrices and input, state and memory cells using the differentiation of matrix vector products described in Appendix A.1.1. If multiple gradients for the same variable are computed, these have to be summed.

A.2 Notation

e	target word
f	source word
I	length of target sentence
J	length of source sentence
i	target sentence index
j	source sentence index
e_1^I	target sentence
f_1^J	source sentence
E	set of possible sentences in the target language
$\hat{e}_1^{\hat{I}}$	hypothesis in the target language
$\Pr(\cdot\|\cdot)$	general probability model
δ	Kronecker delta
a	alignment of a source word to the target sentence
ϕ_i	fertility of a target word with index i
$\boldsymbol{x}^{(n)}$	neurons in layer n of a neural network (NN)

$\boldsymbol{x}^{(n,t=t')}$	values of the neurons in layer n of an NN at time step t'
$\boldsymbol{b}^{(n)}$	biases in layer n of an NN
$\boldsymbol{W}^{(n)}$	weight matrix in layer n of an NN
$\boldsymbol{y}^{(t=t')}$	output of an NN at time step t'
$f(\cdot)$	arbitrary function
$\boldsymbol{c}_t$	memory cell of an LSTM
$\tilde{\boldsymbol{c}}_t$	input candidate of an LSTM at time step t
$\boldsymbol{o}_t$	output gate of an LSTM at time step t
$\boldsymbol{i}_t$	input gate of an LSTM at time step t
$\boldsymbol{f}_t$	forget gate of an LSTM at time step t
$\boldsymbol{a}_t$	state of an LSTM or gated recurrent unit (GRU) at time step t
$\boldsymbol{r}_t$	reset gate of a GRU
t, t', (t, t')	time steps
$\mathcal{L}$	loss function
w_t	weight, part of a weight matrix, at time step t
Δw_t	weight update at time step t
α	learning rate
β	decay rate
$\boldsymbol{f}_j$	one-hot vector indicating the position of word f_j in the source vocabulary
$\boldsymbol{e}_j$	one-hot vector indicating the position of word e_j in the target vocabulary
$\overrightarrow{\boldsymbol{h}}_j$	state of an RNN applied in forward direction
$\overleftarrow{\boldsymbol{h}}_j$	state of an RNN applied in backward direction
$\boldsymbol{h}_j$	source representation of position j
$\tilde{e}_{j,i}$	energy for source position j at time step i
$\alpha_{j,i}$	attention weight for source position j at time step i
$\boldsymbol{c}_i$	context vector of an attention model at time step i

A.3 Corpus statistics

A.3.1 WMT 2017 German→English and English→German

Table A.2: Corpus statistics for WMT 2017 German→English and English→German. In these statistics, the newstest sets from 2008-2014 are not appended to the training set.

	English	German
vocabulary	24,259	24,262
training		
sequences	4,590,101	4,590,101
running symbols	141,126,672	148,809,604
running words	120,554,250	113,881,220
newstest15		
sequences	2,169	2,169
running symbols	58,633	62,562
running words	47,565	44,863
newstest16		
sequences	2,999	2,999
running symbols	80,459	88,302
running words	65,629	64,359
newstest17		
sequences	3,004	3,004
running symbols	80,982	87,345
running words	66,043	62,157

A.3.2 IWSLT 2013 Indomain German→English

Table A.3: Corpus statistics for IWSLT 2013 Indomain German→English

	English	German
vocabulary	11,925	15,833
training		
sequences	138,499	138,499
running symbols	3,019,134	3,076,317
running words	2,763,534	2,607,154
development		
sequences	887	887
running symbols	-	23,094
running words	20,258	19,276

List of Figures

List of Tables

Glossary

Bleu bilingual evaluation understudy (score). 10–12, 25, 55–60, 63–68

Ppl perplexity. 10, 11, 25, 55, 57–60, 62–65

Ter translation edit rate. 10–12, 25, 55–60, 63–68

1D one-dimensional. 2, 3, 18, 20, 42, 54, 56–61, 65–68, 83, 85

2D two-dimensional. 3, 20, 21, 43, 48–50, 58, 61, 64, 65, 67, 83

2D seq2seq 2D sequence to sequence. 48, 49, 63–68, 83, 85

2DLSTM two-dimensional LSTM. v, 1, 2, 19–22, 27, 40, 43–49, 54, 57–65, 67–69, 73, 83, 85

Adam adaptive moment estimation. 29, 31

ANN artificial neural network. 3, 13, 51

ASR automatic speech recognition. 2

BPE byte pair encoding. 52–54, 85

BPTT backpropagation through time. 16, 27

CEC constant error carousel. 17

CNN convolutional neural network. 33

CPU central processing unit. 22

DNN deep neural network. 15, 16

FFNN feedforward neural network. 14–16, 24, 33

GPU graphics processing unit. 22

GRU gated recurrent unit. v, 2, 23, 40, 41, 56–58, 79, 85

IWSLT international workshop on spoken language translation. 54

LM language model. 6, 7, 9, 33

LSTM long short-term memory. v, 2, 17–21, 23, 27, 33–37, 40–42, 47, 48, 54, 56–61, 66, 69, 70, 73, 79, 83, 85

MDLSTM multi-dimensional LSTM. 20, 22

MRT minimum risk training. 25

MT machine translation. v, 1–3, 5, 6, 10, 24, 33

NMT neural machine translation. v, 1–3, 21, 33, 37

NN neural network. v, 1, 25, 33, 39, 52, 61, 78, 79

OOV out of vocabulary. 51, 52

pp percent point. 55, 56, 58, 59, 63–65, 67, 68

RETURNN RWTH extensible training framework for universal recurrent neural networks. 2, 22, 45, 47

RNN recurrent neural network. v, 1–3, 16, 17, 20, 23, 24, 26, 41–43, 54, 56–58, 67, 68, 79, 83, 85

SGD stochastic gradient descent. 27–29

SMT statistical machine translation. 2, 3, 6, 8, 9, 53, 83

TED technology, entertainment, design. 54

WMT workshop on machine translation. 54–60, 62–65, 80

Bibliography

P. Bahar, T. Alkhouli, J.-T. Peter, C. J.-S. Brix, and H. Ney. Empirical Investigation of Optimization Algorithms in Neural Machine Translation. *The Prague Bulletin of Mathematical Linguistics (PBML)*, 108(1):13–25, May 2017. ISSN 1804-0462. doi: 10.1515/pralin-2017-0005. URL `https://publications.rwth-aachen.de/record/715053`.

D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *Computing Research Repository (CoRR)*, abs/1409.0473, 2014. URL `http://arxiv.org/abs/1409.0473`.

T. Bayes. An Essay towards Solving a Problem in the Doctrine of Chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763.

Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3:1137–1155, March 2003. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=944919.944966`.

P. F. Brown, J. Cocke, S. A. D. Pietra, V. J. D. Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A Statistical Approach to Machine Translation. *Computational Linguistics*, 16(2):79–85, June 1990. ISSN 0891-2017. URL `http://dl.acm.org/citation.cfm?id=92858.92860`.

P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics*, 19(2):263–311, June 1993. ISSN 0891-2017. URL `http://dl.acm.org/citation.cfm?id=972470.972474`.

C. Callison-Burch, M. Osborne, and P. Koehn. Re-evaluating the Role of BLEU in Machine Translation Research. In *European Chapter of the Association for Computational Linguistics (EACL)*, pages 249–256, 2006.

S. F. Chen and H. Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics. doi: 10.3115/981863.981904. URL `https://doi.org/10.3115/981863.981904`.

K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D14-1179`.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014.

A. Coates, A. Ng, and H. Lee. An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15, pages 215–223, Fort Lauderdale, FL, USA, 11–13 Apr. 2011. Proceedings of Machine Learning Research (PMLR). URL `http://proceedings.mlr.press/v15/coates11a.html`.

M. R. Costa-Jussà, M. Farrús, J. B. Mariño, and J. A. R. Fonollosa. Study and Comparison of Rule-Based and Statistical Catalan-Spanish Machine Translation Systems. *Computing and Informatics*, 31(2):245–270, 2012. URL `http://www.cai.sk/ojs/index.php/cai/article/view/940`.

D. Coughlin. Correlating Automated and Human Assessments of Machine Translation Quality. In *Proceedings of Machine Translation Summit IX*, pages 63–70, 2003.

G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989. ISSN 1435-568X. doi: 10.1007/BF02551274. URL `https://doi.org/10.1007/BF02551274`.

P. Doetsch, A. Zeyer, P. Voigtlaender, I. Kulikov, R. Schlüter, and H. Ney. RETURNN: The RWTH Extensible Training framework for Universal Recurrent Neural Networks. *Computing Research Repository (CoRR)*, abs/1608.00895, 2016. URL `http://arxiv.org/abs/1608.00895`.

P. Gage. A New Algorithm for Data Compression. *C Users Journal*, 12(2):23–38, February 1994. ISSN 0898-9788. URL `http://dl.acm.org/citation.cfm?id=177910.177914`.

F. A. Gers and J. Schmidhuber. Recurrent Nets that Time and Count. In *IEEE International Joint Conference on Neural Networks (IJCNN)*, volume 3, pages 189–194, 2000. URL `http://dblp.uni-trier.de/db/conf/ijcnn/ijcnn2000-3.html#GersS00`.

F. A. Gers, J. A. Schmidhuber, and F. A. Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, October 2000. ISSN 0899-7667. doi: 10.1162/089976600300015015. URL `http://dx.doi.org/10.1162/089976600300015015`.

X. Glorot and Y. Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL `http://proceedings.mlr.press/v9/glorot10a.html`.

I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout Networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML)*, volume 28, pages III–1319–III–1327. JMLR.org, 2013. URL `http://dl.acm.org/citation.cfm?id=3042817.3043084`.

A. Graves. Generating Sequences with Recurrent Neural Networks. *Computing Research Repository (CoRR)*, abs/1308.0850, 2013. URL `http://arxiv.org/abs/1308.0850`.

A. Graves and J. Schmidhuber. Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks. In *Proceedings of the 21st International Conference on Neural Information Processing Systems (NIPS)*, pages 545–552, USA, 2008. Curran Associates Inc. ISBN 978-1-6056-0-949-2. URL `http://dl.acm.org/citation.cfm?id=2981780.2981848`.

A. Graves, S. Fernández, and J. Schmidhuber. Multi-dimensional Recurrent Neural Networks. In J. Marques de Sá, L. A. Alexandre, W. Duch, and D. P. Mandic, editors, *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, Proceedings, Part I*, volume 4668 of *Lecture Notes in Computer Science*, pages 549–558. Springer, 9–13 Sept. 2007. ISBN 978-3-540-74689-8. doi: 10.1007/978-3-540-74690-4_56. URL `https://doi.org/10.1007/978-3-540-74690-4_56`.

S. Hashem and B. Schmeiser. Improving Model Accuracy using Optimal Linear Combinations of Trained Neural Networks. *IEEE Transactions on Neural Networks*, 6(3):792–794, May 1995. ISSN 1045-9227. doi: 10.1109/72.377990.

S. Hochreiter. Untersuchungen zu Dynamischen Neuronalen Netzen. Diploma thesis, Computer Science Department, Prof. Brauer, Technische Universität München, 1991.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `http://dx.doi.org/10.1162/neco.1997.9.8.1735`.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-term Dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL `http://www.sciencedirect.com/science/article/pii/0893608089900208`.

M. J.-D., T. Dwojak, and R. Sennrich. The AMU-UEDIN Submission to the WMT16 News Translation Task: Attention-based NMT Models as Feature Functions in Phrase-based SMT. *Computing Research Repository (CoRR)*, abs/1605.04809, 2016. URL `http://arxiv.org/abs/1605.04809`.

N. Kalchbrenner and P. Blunsom. Recurrent Continuous Translation Models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Seattle, October 2013. Association for Computational Linguistics.

N. Kalchbrenner, I. Danihelka, and A. Graves. Grid Long Short-Term Memory. *Computing Research Repository (CoRR)*, abs/1507.01526, 2015. URL `http://arxiv.org/abs/1507.01526`.

D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *Computing Research Repository (CoRR)*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

R. Kneser and H. Ney. Improved Backing-off for M-gram Language Modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184 vol.1, May 1995. doi: 10.1109/ICASSP.1995.479394.

V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

J. Li, A. Mohamed, G. Zweig, and Y. Gong. Exploring Multidimensional LSTMs for Large Vocabulary ASR. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China*, pages 4940–4944. IEEE, 20–25 March 2016. ISBN 978-1-4799-9988-0. doi: 10.1109/ICASSP.2016.7472617. URL `https://doi.org/10.1109/ICASSP.2016.7472617`.

F. J. Och and H. Ney. Discriminative Training and Maximum Entropy Models for Statistical Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 295–302, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073133. URL https://doi.org/10.3115/1073083.1073133.

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://doi.org/10.3115/1073083.1073135.

M.'A. Ranzato, S. Chopra, M. Auli, and W. Zaremba. Sequence Level Training with Recurrent Neural Networks. *Computing Research Repository (CoRR)*, abs/1511.06732, 2015. URL http://arxiv.org/abs/1511.06732.

S. J. Reddi, S. Kale, and S. Kumar. On the Convergence of Adam and Beyond. *International Conference on Learning Representations*, 01, 2018. URL https://openreview.net/forum?id=ryQu7f-RZ. accepted as oral presentation.

F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL https://books.google.de/books?id=P_XGPgAACAAJ.

S. Ruder. An Overview of Gradient Descent Optimization Algorithms. *Computing Research Repository (CoRR)*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. volume 1, chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X. URL http://dl.acm.org/citation.cfm?id=104279.104293.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of Research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6. URL http://dl.acm.org/citation.cfm?id=65669.104451.

T. N. Sainath and B. Li. Modeling Time-Frequency Patterns with LSTM vs. Convolutional Architectures for LVCSR Tasks. In N. Morgan, editor, *Interspeech 2016, 17th Annual Conference of the International Speech Communication Association, San Francisco, CA, USA*, pages 813–817. ISCA, 8–12 Sept. 2016. doi: 10.21437/Interspeech.2016-84. URL https://doi.org/10.21437/Interspeech.2016-84.

R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. *Computing Research Repository (CoRR)*, abs/1508.07909, 2015. URL `http://arxiv.org/abs/1508.07909`.

S. Shen, Y. Cheng, Z. He, W. He, H. Wu, M. Sun, and Y. Liu. Minimum Risk Training for Neural Machine Translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*. The Association for Computer Linguistics, 7–12 Aug. 2016. URL `http://aclweb.org/anthology/P/P16/P16-1159.pdf`.

G. F. Simons and C. D. Fennig. *Ethnologue: Languages of the World*, volume 20. sil International Dallas, TX, 2017.

M. Snover, B. Dorr, R. Schwartz, L. Micciulla, and J. Makhoul. A Study of Translation Edit Rate with Targeted Human Annotation. In *Proceedings of Association for Machine Translation in the Americas*, pages 223–231, 2006.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`.

R. S. Sutton. Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986.

Theano Development Team. Theano: A Python Framework for fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

J. Utans. Weight Averaging for Neural Networks and Local Resampling Schemes. In *Proc. AAAI-96 Workshop on Integrating Multiple Learned Models*, pages 133–138. AAAI Press, 1996.

B. van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. W.-F., K. Chorowski, and Y. Bengio. Blocks and Fuel: Frameworks for Deep Learning. *Computing Research Repository (CoRR)*, abs/1506.00619, 2015. URL `http://arxiv.org/abs/1506.00619`.

P. Voigtlaender, P. Doetsch, and H. Ney. Handwriting Recognition with Large Multidimensional Long Short-Term Memory Recurrent Neural Networks. In *International Conference on Frontiers in Handwriting Recognition*, pages 228–233, Shenzhen, China, October 2016. IAPR Best Student Paper Award.

B. Zhang, D. Xiong, and J. Su. Recurrent Neural Machine Translation. *Computing Research Repository (CoRR)*, abs/1607.08725, 2016. URL `http://arxiv.org/abs/1607.08725`.

B. Zhang, D. Xiong, J. Su, and H. Duan. A Context-Aware Recurrent Encoder for Neural Machine Translation. *IEEE/ACM Trans. Audio, Speech & Language Processing*, 25(12):2424–2432, 2017. doi: 10.1109/TASLP.2017.2751420. URL `https://doi.org/10.1109/TASLP.2017.2751420`.