Master of Science (M.Sc.)
M.Sc. in Computer Science and Engineering
M.Sc. in Mathematical Modelling and Compute

DTU Compute
Department of Applied Mathematics and Computer Science

Character-level Machine Translation

Neural Machine Translation

Alexander Rosenberg Johansen (s145706@student.dtu.dk)
Elias Khazen Obeid (s142952@student.dtu.dk)
Jonas Meinertz Hansen (s142957@student.dtu.dk)

Supervisor and Co-supervisor:
Ole Winther (olwi@compute.dtu.dk)
Casper Kaae Sønderby (casperkaae@gmail.com)

Kongens Lyngby
July 16th 2016

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In this report we document our efforts to build a program which employs recurrent neural network methods for making accurate and natural translations from one language to another.

We present and explain the general theory behind recurrent neural networks and how to apply it in the context of natural language translation. We explain general architecture and the interesting details of our concrete and flexible software implementations of the methods, which has enabled us to quickly and easily test tweaks and variations in the models.

Among our novel contribution are a variable bucket schedule, which allows us to make better use of the available RAM and train on more data in the same amount of time. We also employ a clever word segmentation setup that allow our model to take individual characters as input while having a recurrent neural network that only needs to keep track of the overall context on the word-level.

We built our models with the TensorFlow machine learning framework. Our final models achieve close to state-of-the-art performance, e.g. reaching a BLEU score of 17.43 on the WMT '15 En–De dataset.

Preface

This report documents the master thesis in Computer Science & Engineering and Mathematical Modelling at the Technical University of Denmark. Each of the report's sections will have a responsible author, which will be apparent right below the section title. Keep in mind that the group members have worked in collaboration throughout the project to compose this report.

The report is divided into six chapters; chapter 1 which introduces the project and the rest of the report, chapter 2 that gives relevant background knowledge, chapter 3 where we present and describe different tools we used and which considerations we have made, and chapters 4 and 5 present our implementation and the various experiments performed and results gained, lastly chapter 6 presents our conclusion and thoughts for future work. The end of the report consists of various appendices and a bibliography.

Kongens Lyngby, July 16, 2016

Alexander Rosenberg Johansen (s145706@student.dtu.dk)
Elias Khazen Obeid (s142952@student.dtu.dk)
Jonas Meinertz Hansen (s142957@student.dtu.dk)

Contents

CHAPTER 1

Introduction

This report documents the master thesis of Alexander, Elias, and Jonas in Computer Science & Engineering and Mathematical Modelling at the Technical University of Denmark. The project stretched from the 1st of February to the 16th of July 2016. Our code base developed throughout this project is publicly available[1].

In this project we utilise learning algorithms to translate from one language to another. Throughout the report we refer to learning algorithms also as models and computational graphs, interchangeably. This will mostly depend on the meaning of the paragraph in which it is used. Typically, learning algorithms for computers are categorised as machine learning algorithms. The algorithms investigated in this project are a specific branch of machine learning known as deep learning. More specifically, we work with neural machine translation because our learning algorithms will be based on neural networks. [Goodfellow et al., 2016]

To better understand the relevant machine learning topics, we present background knowledge and machine learning examples in chapter 2. Chapter 3 presents our considerations together with tools that were relevant for the project. In chapter 4 we present the implementation details of our project. With the implementation we perform various experiments to try different ideas and configurations for our models. These experiments are presented with results in chapter 5. Lastly, we present our conclusion in chapter 6 together with a future work and acknowledgements.

The ensuing sections of this chapter will begin with a general explanation of what learning algorithms are (section 1.1) followed by a literature review on the topic of machine translation (section 1.2). Afterwards, we motivate our work in section 1.3 and continue to present our problem statement in section 1.4. Finally, in section 1.5 we conventions and various mathematical notation used throughout the report.

1.1 Learning Algorithms

Responsible: Elias Obeid (s142952)

Machine learning algorithms are algorithms that are capable of learning from a given set of structured data points. A more general definition of learning algorithms, and what precisely is meant by "learning", is supplied by Mitchell [1997]:

[1] `https://github.com/styrke/master-code`

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

This definition applies emphasis on finding and modelling a task in which experience can improve the algorithm's performance for that task.

1.1.1 The Task

With machine learning it is possible to tackle a wide range of tasks that are very hard to solve with explicitly hand crafted algorithms. Clever as they are these methods don't provide us with any means of circumventing the challenge stated by the "no free lunch" theorems, which by the words of their authors, Wolpert [1996], Wolpert and Macready [1997], demonstrate that:

> [...] If an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems.

However, we do get to choose the class of problems that we want the algorithm to perform well on, by being explicit about *the task* that we pick and optimise the algorithm to solve.

We define the task that we are focusing on in this report in section 1.4.

1.1.2 The Performance Measure

The performance measure generally serves as a formal objective measure of how close the algorithm is to solving *the task* perfectly.

Choosing a good performance measure is not always trivial. For various reasons, it may not be possible to optimise with respect to the ideal performance measure in which case it is necessary to chose another performance metric that hopefully correlates well with the ideal. However, small subtle differences in the performance metrics can potentially have a significant impact on how well the final algorithm solves the task.

For translation we would ideally utilise the BLEU score (described in section 2.9.2), which was specifically developed for measuring the accuracy and quality of translated text, and is widely used when reporting results for machine translation [Papineni et al., 2002a]. Unfortunately, the BLEU score is not continuous, and thus cannot be used with the optimisation algorithms we're using for our networks. As our task deals with sequences of symbols, we use the perplexity measure instead which, while not being exactly the metric we care about, manages to give good BLEU scores as well.

1.1.3 The Experience

Training machine learning algorithms to solve tasks, typically happens by "showing" a set of data points to the algorithm that it can learn to perform the given task on. In

some sense this can be thought of as the machine learning algorithm *experiencing* the data like a chess player experiences a lot of chess playing in order to become a grand master. Such experience allows the machine learning algorithm to mold the function such that it minimises a given performance metric. [Goodfellow et al., 2016]

1.2 Literature Review

Responsible: Alexander Johansen (s145706)

Considerable progress has been made in the field of natural language processing in recent years. In the following we describe previous work on neural machine translation which is relevant background knowledge and basis for this report.

Translating text between languages is a central task in modern Natural Language Processing (NLP). Machine Translation (MT) refers to the use of software to perform such translations. From a probabilistic perspective, MT is equivalent to finding a sentence $\mathbf{y}$ that maximises the conditional probability of $\mathbf{y}$ given a source sentence $\mathbf{x}$. Neural Machine Translation (NMT) is the application of parametrised models to learn the conditional probability used in MT. State of the art (SotA) NMT has recently matched the best known MT systems [Sutskever et al., 2014].

The SotA approach for NMT is a task currently performed by using an encoder-decoder architecture, which in broader contexts is also known as a sequence-to-sequence model [Cho et al., 2014c, Sutskever et al., 2014]. This model is often parameterised by using two Recurrent Neural Networks (RNNs) — one for the encoder and one for the decoder, and is thus known as a *RNN encoder-decoder* model [Cho et al., 2014c, Sutskever et al., 2014]. For translation, such models often rely on encoding and decoding whole words at a time, and are also referred to as word-to-word models. To mention one challenge with word level translation, it is that new and unknown words are often translated by simply copying the word or adding a special character that represent any word was not recognised.

Variants of the RNN encoder-decoder model with attention have significantly improved on translation performances compared to a regular RNN encoder-decoder model. The model's attention mechanism attempts to highlight the sets of words from the given input sequence that are most relevant to the decoder when it predicts each word in the output sentence. [Bahdanau et al., 2014]

To adapt the RNN encoder-decoder model to a character based approach Chung et al. [2016] presents a model where they encode and decode combinations of characters that are most likely to appear in the data. This combination of characters is similar to Byte-Pair Encoding (BPE) where the pairs are of characters instead of bytes [Witten et al., 1999], and is therefore referred to as a bpe-to-char model. Thus, the encoder uses the BPE as input, while the decoder outputs a sequence of characters. The idea is that a subset of the characters in the words are often reused in the construction of other words [Chung et al., 2016]. Moreover, Ling et al. [2015b] presents a character based adaptation of a word-to-word model. The model expands the RNN encoder-decoder

attention variant with word pre-training, i.e. a character-to-word mechanism on the encoder side and a vector-to-character mechanism on the decoder side.

1.3 Motivation

Responsible: Jonas Hansen (s142957)

The fields of machine learning and deep learning are evolving rapidly. The steep increase in available computing power during the last decade, due to the appearance of generally programmable GPUs, has finally made it feasible to utilize the techniques in many new areas.

This has lead to much faster and more encouraging progress in many areas than most observers would have predicted before seeing it. The error rate for the winning entries in the yearly ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has decreased drastically over the years (see fig. 1.1) and is now in some senses competitive with humans in speed (and certainly in reliability). A top-level professional go-player was recently defeated by a computer program powered by neural networks [Silver et al., 2016]. Furthermore, our phones can transcribe and react to what we say.

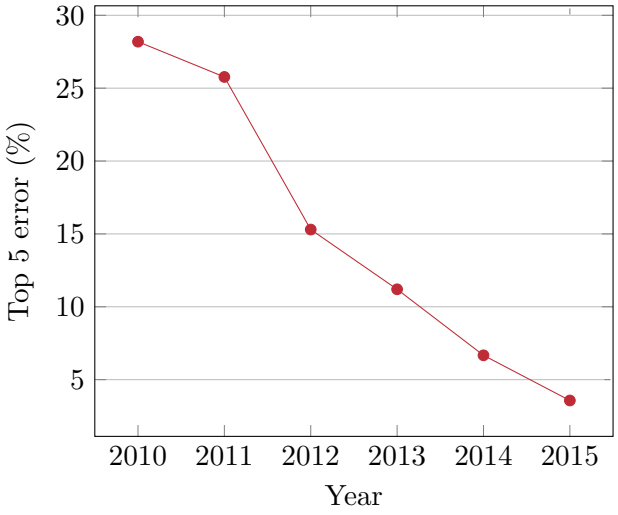

Figure 1.1: Top 5 error rate for the winning solution for ILSVRC by year. Top 5 error means that having the correct label among the top 5 most likely predictions counts as a correct classification.

The experiences and enthusiasm gained from these successes are carrying over to many other applications including natural language processing. In the domain of natural language processing, neural networks can now among other things be used to

- Search, rank, summarise, and translate text
- Describing pictures with natural sentences [Karpathy and Fei-Fei, 2015]

- Answer basic questions based on a set of stated facts [Kumar et al., 2015, Peng et al., 2015]
- Part-of-speech tagging, dependency parsing and sentence compression [Andor et al., 2016]
- Propose email responses based on the content of a message[2]

It is possible to build very large scale systems that apply these techniques for real world applications[3].

We want to be part of the gold rush in this exciting field, by replicating the state-of-the-art in character based neural machine translation, and possibly make our own contributions.

Translation is a very large problem within the field of Natural Language Processing. Most people only know one or two languages well enough to use them effortlessly in their daily lives, rendering them unable to use any information that is recorded in other languages than those that they understand well. With almost 7000 living languages [Anderson, 2010] people are prevented from communicating with most other people and accessing most of the world's recorded information.

Manual translation does not scale very well. In order to make twice as many translations, twice the effort will have to be put in. In comparison machine translation scales very well, but is generally wanting in quality. We hope to contribute even a tiny amount of progress towards a world where all information is generally accessible to and usable by everyone.

1.4 Problem Statement

Responsible: Elias Obeid (s142952)

The goal with the project is to explore, and possibly improve, the state of the art for character based neural machine translation.

Essentially we want to use deep learning methods to make a computer program that solves the following informally defined task:

> Take a sequence of input symbols from a source language, and predict the sequence of symbols in a target language, that most natural sounding and accurately conveys the meaning of the source sentence.

This includes, but is not limited to, investigating modern methods, implementing a program for training and using models, and writing this report documenting it all.

More specifically, the project's aim is to investigate the use of characters instead of words as symbols. Previously, words have been the primary driver behind the success of Natural

[2] `https://research.googleblog.com/2015/11/computer-respond-to-this-email.html`

[3] `https://code.facebook.com/posts/181565595577955/introducing-deeptext-facebook-s-text-understanding-engine/`

Machine Translation. [Cho et al., 2014c, Bahdanau et al., 2014, Sutskever et al., 2014, Luong et al., 2015a] However, recently character- and Byte-Pair Encoding based models have been demonstrated to yield equivalent, if not better, translation performance than word based models. [Ling et al., 2015b, Chung et al., 2016]

1.4.1 Working with Characters instead of Words

As described in the literature review section, (see section 1.2) most of state of the art neural machine translation models rely on word level encoding, which means that the model must be fed with a whole word at a time. This approach has the following disadvantages:

1. Each word must be tokenized (converted to an id that represents the word) before it can be fed into the model. Because each word that the program is able to understand must have its own unique id, the total number of words that a model can differentiate between is limited by computational capacity. Usually this means that only the k most frequent words in the data set are included in the model's vocabulary. The remaining words are treated as completely unknown.
2. Different forms of the same word are presented as completely different tokens when they are fed to the model. Depending on the preprocessing that is used, even different capitalizations of the same word may seem completely unrelated.
3. The model has no ability to take words that are not in the vocabulary into consideration, even if they are just different forms or misspellings of known words.

These, and some other problems, can be avoided by using character based models that consider each character individually.

Character based models on the other hand has some other challenges to overcome chiefly among them a character based model has to work with sequences that are much longer.

In our first experiment we build a deep learning model based on previous word level models, with the purpose of modifying it to work on character level instead. Moving to characters will reduce the required number of input tokens to only a few hundred in the Romance languages[4]. However, this does present a number of challenges, such as longer input and output sequences (an English word is on average 5.1 characters long). Moreover, as the algorithm is fed a single character at a time it must also be fed with space character. These indicate the end and beginning of words and we must consider how to handle spaces properly. This more basic representation of a sentence should contain more information than a tokenized word would.

We believe that a character based model will be able to generalise better to previously unobserved words because even unknown words may have some sub-sequence of characters in common with a known word which the model will be able to detect and exploit.

[4]The Romance (or Latin) languages are the modern languages that evolved from Vulgar-Latin between the sixth and ninth centuries and formed the Italian and Indo-European language family. Today these languages still share most of the characters in their alphabets.

1.5 Notation and Conventions

Responsible: Alexander Johansen (s145706)

Throughout the report we will adhere by the following notation and convections. Variables representing scalar values will be defined as lower cased letters, i.e. s is a scalar.

Vectors will be defined as lower case bold letters, i.e.

$$\mathbf{v}^\top = (v_1, v_2, \ldots, v_n) \tag{1.1}$$

is a column vector of length n. Often when talking about a sequence or vector $\mathbf{x}$, we will let T_x denote the length of the sequence unless something else is implied by the context.

Matrices will be written with bold upper case letters, i.e. $\mathbf{M}$ is a matrix. Matrices may be defined as $\mathbf{M} \in \mathbb{R}^{n \times m}$, which defines the matrix $\mathbf{M}$ to have n rows and m columns.

Throughout the report we will refer to *tensors* of some rank. Tensors may be of arbitrary rank, where the rank defines the dimensions the tensor has. A tensor of rank 2 (or a two-dimensional tensor) is for instance a matrix.

CHAPTER 2

Methods

This chapter presents relevant background knowledge and simple examples of machine learning algorithms and their applicability. Initially, we provide a general description of machine learning basics in section 2.1, followed by some basic examples and principles (Linear- and Logistic Regression) in sections 2.2 and 2.3. We present the basic workings of an Artificial Neuron (section 2.4) followed by a description of Artificial Neural Networks (section 2.5). In section 2.6 we present the Backpropagation algorithm (section 2.6.1) and some relevant optimisation algorithms (section 2.6.2). Moreover, we introduce Recurrent Neural Networks (section 2.7) wherein we introduce Gated Recurrent Units (section 2.7.2) and the Encoder-Decoder architecture with the Attention mechanism (section 2.7.4). At the end we introduce the concept of embeddings in section 2.8 and how translation performance is measured in section 2.9.

2.1 Basics of Machine Learning

Responsible: Jonas Hansen (s142957)

Machine learning is a subfield of Computer Science that uses computational statistics for pattern recognition to solve a given task. A task could for instance be to organize digital photos by their content, transcribe an audio clip, recommend new media based on preferences, etc. Machine learning tasks are often divided into two overarching paradigms; *unsupervised* and *supervised*. These two paradigms do not have any formal definition and their methods and uses can often overlap. [Goodfellow et al., 2016]

- Unsupervised models are centred around finding an underlying structure in a given dataset. This is often achieved by obtaining (an approximation of) the probability distribution $p(x)$ that generated the dataset x. A classical example of unsupervised learning is clustering, where the goal is to organise unlabelled samples of data into groups/categories based on their relative locations in the feature space. Another example could be finding the probability function of properties/features of some residence (e.g. learning the relation between area m^2 and number of rooms, or location, etc.).
- Supervised models are centred around finding conditional probability distributions, $p(y \mid x)$, over a given domain. Such that given a random variable x the machine learning model could determine the dependant variable y. An example of supervised learning could be that a model has learned the conditional probability of the price of a house given its real estate properties such as area, number of rooms, and

location. Another example is categorising the motive of an image when given the raw pixel data as input.

2.1.1 Machine Learning Models

When we have chosen a task to work with, we essentially desire to find a function, f, that can solve that task. For example if the task is to decide whether or not there are cats in pictures, we might want a function that takes a picture as input and returns a 1 (true) if there is a cat in the input picture, and 0 (false) otherwise. This is a tall order, and rarely possible to do perfectly. For any possible task that we want our program to solve, there is an infinite amount of possible functions to choose from. Each of the possible individual functions can be thought of as a hypothesis about how to solve the task when given a sample, and most of them are hopelessly inadequate for solving the task. So we make some simplifications to help us choose an appropriate function among all the possibilities.

Firstly, instead of considering *all* functions that have the desired domain and codomain, we restrict our search to a *parametrised family of functions*, or a "*model*". We denote the model as f_θ to indicate that its behaviour is controlled by the parameter θ, and we can move between the functions that belong in the family by changing the value of θ.

2.1.1.1 Defining the Model's Performance

Given that we have chosen an appropriate model, we have then limited the scope of our problem to choosing the best value of θ. But how exactly do we then choose the best value of θ? We define a *loss function* (also referred to as cost, error, or objective function) that quantifies exactly how good we consider some prediction to be compared to some target. It is then possible to compare the losses of the model for different values of θ and choose the one that performs best.

The loss function is conventionally defined such that better performance on the task results in lower loss, and worse performance conversely results in higher loss. This convention which fits the casual understanding of the word "loss" as a bad thing that should be avoided or minimised. Thus, the loss function tells the model how well it is doing on solving the task.

By choosing the value of θ that minimises the loss function, we would simultaneously find the function (from our family) that best performs the task according to the loss function, when given the training data.

2.1.1.2 Training the Model on Appropriate Data

It is important that the training data is representative of the data that the model will actually perform its task on, as this will also affect the chosen value of θ. The goal with machine learning is to develop models that solve given tasks generally well when given data that is similar to the data on which it was trained. This includes data that the

model has never seen before. One concern is that the model becomes so good at making its predictions on the training data that its performance suffers on unseen data. This phenomenon is called *overfitting* because it is caused by the model fitting the training data too well.

The first step to combating this challenge is to detect it. Thus it is customary to split the dataset into disjointed subsets dedicated to training, validation (sometimes referred to as "development"), and testing. The model is then trained exclusively on the appropriately named training set, and regularly during training the model's performance is evaluated on the validation set. When the model has processed the entire data set one says that one *epoch* has passed, and many epochs may pass before the training has finished. Because the model has basically never seen the validation split, the performance on the validation set can be used to assess how well the model generalises to novel data at different points during the training. If a model is overfitting the training data, there are a few general possibilities for reducing it:

- use more training data
- introduce artificial noise in the data or directly in the model
- increase bias by e.g. using *weight decay*, which is an extra term that penalizes the model for using parameters that are far from zero

Bias and Variance Trade-off. Introducing more *bias* to the model is a way to constrain the model's complexity, which may cause the model to be unable to find relevant relations in the training data and the expected output. In some sense this is the opposite of overfitting and is called *underfitting*, because the model is unable to actually represent the data which is more complex than the model is allowed to be. Too much bias is not good for a model, and this is where *variance* enters the picture. A model is said to have higher variance if it is less constrained and may become more complex. Thus, with too much variance the model may overfit and try to represent noise in the data instead of the actual features. It is important to be able to balance the models in a way to gain the best generalisation.

Assessing the Model's Generalisation. It is important that the model is not also trained on the validation set, as it will then be prone to directly overfitting the validation data as well. However, when the validation set has been used for making decisions indirectly about the model, the model may in some sense end up overfitting the validation set anyway. The performance on the validation set will for example often be used to make decisions about the details of the architecture of the model (the hyperparameters) by choosing the values or the architecture that gives the best performance on the validation set. For this reason the test set is used as a final assessment of how well the model will generalise on unseen data.

2.2 Linear Regression

Responsible: Elias Obeid (s142952)

Linear regression is a supervised machine learning algorithm. It maps a given input vector of size m to a continuous scalar value, i.e. $f : \mathbb{R}^m \to \mathbb{R}$. This mapping associates a set of weights $\mathbf{w}$ (the parameters) to each value in the input vector $\mathbf{x}$ and adds a scalar bias value b. This mapping is often referred to as the hypothesis function and defined as

$$h_w(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b, \tag{2.1}$$

where $\mathbf{w}^\top = (w_0, \ldots, w_m)$ and $\mathbf{x}^\top = (x_0, \ldots, x_m)$ are vectors in $\mathbb{R}^m$, and $b \in \mathbb{R}$ is a scalar. Often when formulating linear regression the bias b is omitted, instead $\mathbf{w}, \mathbf{x}$ are expanded such that they are vectors in $\mathbb{R}^{(m+1)}$, where they are shifted such that $w_0 = b$ and $x_0 = 1.0$.

2.2.1 Measuring the Performance

In order to measure the performance of the hypothesis function a loss function must be chosen. For simplicity we use the mean squared error (MSE) to demonstrate the use of a loss function. MSE gives a measurement of the average "distance" between the model's prediction, $y = h_w(\mathbf{x})$, and the expected target value, t. We defined the hypothesis function $h_w(\mathbf{x})$ w.r.t. the weights given by $\mathbf{w}$ and the input $\mathbf{x}$.

Let us assume that all input vectors are combined in a matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$, i.e. an $n \times m$ matrix. Each row of $\mathbf{X}$ gives one input sample, i.e. the i^{th} row is given by $\mathbf{X}_i$ and yields one input vector. Moreover, $\mathbf{X}_{i,k}$ fetches the k^{th} element from the i^{th} vector which is a scalar. We can now define the loss function L w.r.t. to the weights as the MSE given by

$$L(\mathbf{w}) = \frac{1}{2n} \sum_{i=1}^{n} \left(h_w(\mathbf{X}_i) - \mathbf{t}_i \right)^2, \tag{2.2}$$

where n is the number of samples, and $\mathbf{t}_i$ yields the i^{th} target value. Note that the results is halved for ease of differentiation.

2.2.2 Minimising the Performance Error

Some functions can be minimised with a closed-form solution[1], which is an expression that uses a finite amount of mathematical operations and functions to minimise the solution. Linear regression with the MSE loss function can be minimised with the closed-form solution known as least squares. Moreover, for illustrative purposes we present its optimisation with gradient descent.

Closed-form Solution. The least squares equation finds a set $\mathbf{w}$ that minimizes the loss function, i.e. $\mathbf{w}$ is an optimal solution, as

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}, \tag{2.3}$$

where $\mathbf{X}$ is a matrix as described above.

[1] `https://en.wikipedia.org/wiki/Closed-form_expression`

Optimising with Partial Derivatives. Sometimes the closed-form solution is either too expensive or does not exist at all. Then it is possible to approximate a solution using e.g. gradient descent which uses the partial derivatives to compute how the weights are to be updated to minimise the loss.

A walkthrough of calculating the partial derivatives is presented in appendix A, and we simply present the final result here. We assume we have two parameters in the hypothesis, w_0 and w_1, where w_0 is a bias term as explained earlier. Parameter updates are thus performed using the partial derivatives/gradients w.r.t. the weight/parameter being evaluated.

$$\frac{\partial}{\partial w_1} L(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \left(w_0 + w_1 \mathbf{X}_{i,1} - \mathbf{t}_i \right) \mathbf{X}_{i,1}, \tag{2.4}$$

where $\mathbf{X}_{i,1}$ is the input associated with w_1, and thus $\mathbf{X}_{i,0}$ is simply 1.0 and omitted from the equation. These gradients can now be used with some gradient-based optimiser (section 2.6.2) to update the weights in the aim of minimising the performance error given by the loss function. An example of such an optimiser is gradient descent.

2.3 Logistic Regression

Responsible: Alexander Johansen (s145706)

Logistic regression is an extension of linear regression to handle prediction of discrete values. We introduce binomial logistic regression in section 2.3.1, which leads us to multinomial logistic regression in section 2.3.2.

2.3.1 Binomial Logistic Regression

Binomial logistic regression is a method to handle binary classification. The intuition and mathematics behind binomial logistic regression is very similar to multinomial logistic regression. Thus exploring the binomial version is helpful to understand the multinomial version. The hypothesis function for binomial logistic regression is defined as

$$h_w(\mathbf{x}) = \text{sigmoid}\left(\mathbf{w}^\top \mathbf{x} \right), \tag{2.5}$$

where the sigmoid function (see section 2.4.2) is used to squash the linear combination (ref. logit) to be between zero and one, i.e. $0 < h_w(\mathbf{x}) < 1$. The sigmoid function is illustrated in fig. 2.1.

The general idea of the sigmoid function is to squash the logits into a probability domain and find the prediction of some class/category, e.g. $p(y = 1 \mid \mathbf{x};\ \mathbf{w})$, which reads "the probability of y is 1 given $\mathbf{x}$ parametrised by $\mathbf{w}$".

2.3.1.1 Binomial Cross Entropy

In section 2.2.1 we presented the MSE loss function for linear regression. MSE has been shown to yield poor results with first-order gradient-based optimization methods because output units can saturate leading to vanishing gradients. [Goodfellow et al., 2016]

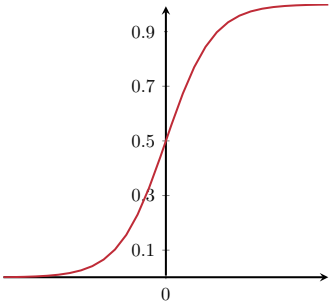

Figure 2.1: Illustrating the sigmoid function, i.e. $\text{sigmoid}(x) = \dfrac{1}{1+e^{-x}}$.

The following formulation is often used as the loss function for logistic regression and known as binomial cross entropy. It is based on the negative log-likelihood which, given the prediction, expresses to which extent the parameters must be updated. Binomial cross entropy it is defined as

$$\text{cost}(\mathbf{x}, t) = \begin{cases} -\log(h_w(\mathbf{x})) & \text{if } t = 1 \\ -\log(1 - h_w(\mathbf{x})) & \text{if } t = 0 \end{cases} \tag{2.6}$$

$$= -t \log\left(h_w(\mathbf{x})\right) - (1-t) \log\left(1 - h_w(\mathbf{x})\right). \tag{2.7}$$

For classification problems, the expected values will always be either 0 or 1. Therefore, it is possible to simplify the function as presented in eq. (2.7).

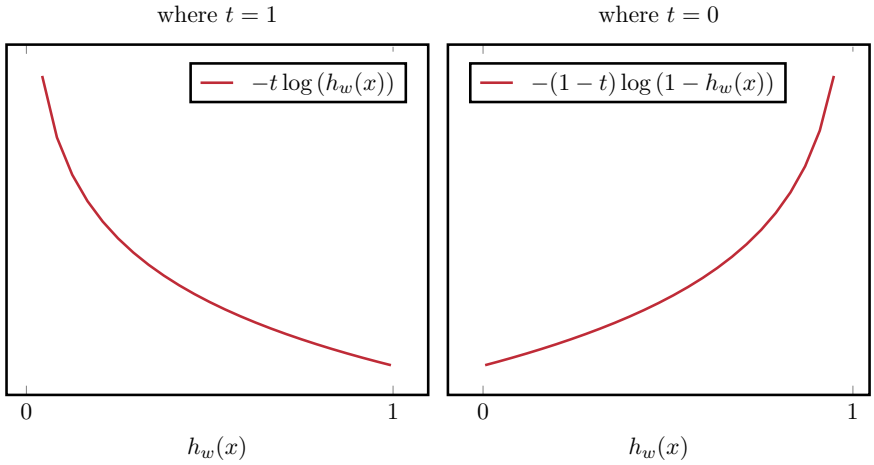

Figure 2.2: The two curves of the logarithms given the prediction – the worse the prediction the greater the penalty.

The intuition behind cross entropy is that it will penalize wrong model predictions such that very confident wrong predictions will be penalized exponentially. To strengthen this intuition we present the plots in fig. 2.2, where we see that if the prediction is not as

expected the cost quickly grows. We now have the entire loss function, $L(\mathbf{w})$, for logistic regression as

$$L(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \text{cost}\left(h_w(\mathbf{X}_i), \mathbf{t}_i\right). \tag{2.8}$$

2.3.2 Multinomial Logistic Regression

Multinomial logistic regression is a method to handle multinomial classification. Such as classifying a symbol amongst multiple possible symbols. The hypothesis function for multinomial regression is defined as

$$h_w(\mathbf{x}) = \text{softmax}(\mathbf{W}^\top \mathbf{x}), \tag{2.9}$$

where the softmax function (see section 2.4.2) squashes the logits of a k-dimensional vector, such that the logits share a probability distribution and sums to one. Note, that the weights are now defined in a matrix $\mathbf{W}$ as we have multiple possible outputs. The softmax function thus becomes a generalisation of the sigmoid function as it finds the probability of a given class in a set of classes and presents the probability of the c^{th} class as $p(y_c = 1 \mid \mathbf{x}; \mathbf{W}_c)$.

2.3.2.1 Multinomial Cross Entropy

To be able to minimise the loss one must compute the negative log-likelihood of the hypothesis function, which in our case is the softmax function. The penalty is thus given by comparing the prediction with the expected target vector as

$$\text{cost}(\mathbf{x}, \mathbf{t}) = \mathbf{t}^\top \big(- \ln(h_w(\mathbf{x}))\big), \tag{2.10}$$

which defines multinomial cross entropy. It penalizes such that the model focusses on the most incorrect prediction (by penalizing exponentially). This means that the model will always focus on examples not yet classified correctly than examples classified correctly, but with low confidence [Goodfellow et al., 2016]. Given a matrix of inputs $\mathbf{X}$ and targets $\mathbf{T}$ the loss function L averages over each data point's error given by eq. (2.11) as

$$L(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} \text{cost}(\mathbf{X}_i, \mathbf{T}_i), \tag{2.11}$$

where n defines the number of data points.

The performance error is thus minimised by updating the weights using the derivatives of the loss- and hypothesis function (see section 2.4.2) with some gradient-based optimisation algorithm (see section 2.6.2).

2.4 Artificial Neuron

Responsible: Jonas Hansen (s142957)

The study of artificial neural networks, or simply neural networks, is a sub-field within machine learning that approximates mappings from an input to labels by transforming the input with several linear combinations and non-linear activations [Nielsen, 2015, Goodfellow et al., 2016]. This section will describe the workings of artificial neurons, which are the building blocks of neural networks, where we introduce the concept of linear separability (section 2.4.1) and present a few relevant activation functions (section 2.4.2).

The fundamental unit of a neural network is the artificial neuron, or simply neuron, which is a basic mathematical model that computes some real valued output given some real valued input vector, i.e. $f : \mathbb{R}^n \to \mathbb{R}$. The output y is computed by applying the input vector $\mathbf{x}$ element-wise to a vector of weights $\mathbf{w}$, and transforming the product with an activation function φ (often, if not always, nonlinear). Given n input values the output can be computed as [Nielsen, 2015, Goodfellow et al., 2016]

$$y = \varphi(\mathbf{w}^\top \mathbf{x}) = \varphi\left(\sum_{i=1}^{n} w_i x_i\right). \tag{2.12}$$

In section 2.4.2 we present some activation functions which are relevant for this project.

2.4.1 Linear Separability

A single neuron can only model simple mathematical functions. To exemplify such, we will consider the boolean OR function, which takes two inputs and yields 1 (true) if at least one of the inputs is 1, otherwise it returns 0 (false). [Goodfellow et al., 2016]

The neuron will receive two inputs (x_1, x_2) which each have a weight (w_1, w_2) associated. Let us define the activation function, $\varphi(z)$, to yield 1 if the weighted sum is greater than or equal to one half, and otherwise it yields 0. We have that z defines the weighted sum and the activation function is defined as

$$\varphi(z) = \begin{cases} 1 & \text{if } z \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

The input values can be either 0 or 1. If we assign both weights to 0.5 the neuron will successfully emulate an OR function. The neuron's output are demonstrated in table 2.3.

The neuron creates a linear classifier that attempts to separate all possible outputs into positive or negative regions. This is however only possible if these can be separated into two regions with a single line.

The exclusive-OR (XOR) function is an example of a function that is not linearly separable, and thus a single neuron is not able to emulate it [Goodfellow et al., 2016]. The XOR function returns 1 if exactly one of the inputs are 1, otherwise it returns 0. Figure 2.4 illustrates that two lines are needed to emulate an XOR function and is thus not

x_1	x_2	z	φ
0	0	0	0
0	1	½	1
1	0	½	1
1	1	1	1

Table 2.3: Results of emulating an OR function.

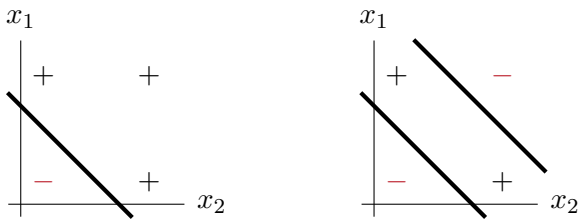

Figure 2.4: Illustrating linear separability, where OR function (left) is linearly separable and XOR function (right) is not.

linearly separable. Each axis defines whether the respective input is 1 or 0. In section 2.5 we demonstrate that a simple neural network is successfully able to emulate an XOR function.

2.4.2 Activation Functions

The neurons transfer information forward in a network. Given some set of input, the neuron's activation function computes the output value to be propagated forward. One can consider the activation function as deciding the presence of some desired feature which the neuron is searching for. If the combined inputs indicate the presence of this feature, the neuron must trigger a positive output value. In the following paragraphs we briefly present functions that are commonly used as activation functions and that are relevant for our project.

Rectified Linear Unit. The Rectified Linear Unit (ReLU) will never output a negative value. [Goodfellow et al., 2016]

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.13}$$

and its derivative is computed as

$$\frac{dy}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.14}$$

Logistic Sigmoid Function. A popular function for binary classification is the logistic sigmoid, or simply sigmoid, which outputs values between 0 and 1. [Goodfellow et al.,

2016]

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}, \tag{2.15}$$

and its derivative is computed as

$$\frac{dy}{dx} = y(1 - y) \tag{2.16}$$

Hyperbolic Tangent Function. Similar to the sigmoid function the hyperbolic tangent has an S-shaped curve and outputs values between -1 and 1. [Goodfellow et al., 2016]

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} \tag{2.17}$$

$$= \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.18}$$

and its derivative is computed as

$$\frac{dy}{dx} = (1 - y^2) \tag{2.19}$$

Softmax Function. The probability distribution over a discrete variable with k possible values is often computed at the last layer with the softmax function. It normalises the probability distribution. [Goodfellow et al., 2016]

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, \quad \text{for } j = 1, \ldots, K. \tag{2.20}$$

The partial derivative of the softmax function is computed for an output y w.r.t. the input x as

$$\frac{\partial y_j}{\partial x_i} = \begin{cases} y_i(1 - y_i) & \text{if } i = j \\ -y_i y_j & \text{if } i \neq j \end{cases} \tag{2.21}$$

These derivatives are used to update the parameters of a network as presented in section 2.6.

2.5 Artificial Neural Network

Responsible: Elias Obeid (s142952)

Artificial Neural Networks, also known as Feedforward Networks or simply Neural Networks (NNs), were inspired by the brain, where a neural network's neurons simulate the brain's axons, the connections between the neurons simulate the dendrites, and the activation function simulates the brain's soma. [Nielsen, 2015, Goodfellow et al., 2016]

A neural network aims to approximate some function f^*. This function could for instance map some input to some category. Consider a classifier $t = f^*(\mathbf{x})$, where $\mathbf{x}$ is the input.

A neural network will define another mapping, $y = f(\mathbf{x};\theta)$, where the values of θ are to be learned in a way to reflect the best approximation of the desired function, f^*, i.e. the aim is to minimise the difference between the expected output t and the predicted output y. [Nielsen, 2015, Goodfellow et al., 2016]

A neural network often consists of multiple connected functions that together aim to compose the approximation f. These functions are associated with a directed acyclic graph that defines how the functions are connected. Say we have three functions f^1, f^2, and f^3 connected as

$$f(\mathbf{x}) = f^3\left(f^2\left(f^1\left(\mathbf{x}\right)\right)\right), \tag{2.22}$$

where each function represents a linear or nonlinear transformation. These functions are also referred to as "layers" because a neural network is often thought of as a graph of layers. [Goodfellow et al., 2016]

The last layer is f^3, also known as the output layer, because it defines the final output of the network. By convention the first layer is the input $\mathbf{x}$, also known as f^0. The intermediary layers, f^1, f^2, are called hidden layers, because the training data does not define in any way what the output of these layers should be to reach an approximation of the desired final result. The training data for supervised learning (as described in section 2.1) is a set of inputs $\mathbf{x}$ and targets $\mathbf{t}$ given by $f^*(\mathbf{x})$.

The depth of a neural network is defined by the number of layers in the function f (including the input- and output layers) [Goodfellow et al., 2016]. The graph that defines f could for instance be as illustrated in fig. 2.5, where an arbitrary number of hidden layers may be connected. Each connecting edge is associated with some value called a weight, which is used to compute the weighted sum of the given layer. These weights/parameters is what the learning algorithm must update while learning.

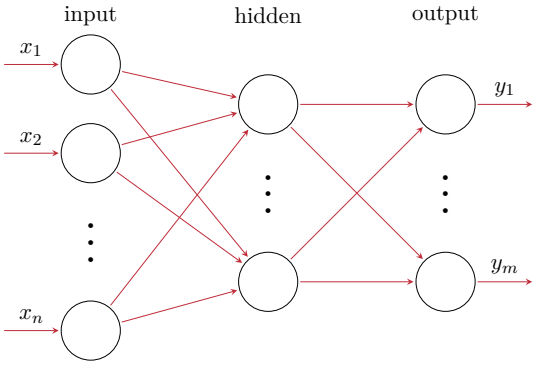

Figure 2.5: Directed acyclic graph illustrating a simple three layered neural network given by $f(\mathbf{x})$.

The weights may be updated using the backpropagation algorithm (see section 2.6.1) with gradient-based optimisation algorithms (see section 2.6.2), such that the neural network learns to best implement an approximation of f^*. [Goodfellow et al., 2016]

The hidden layers are typically represented as vectors and the model's width is determined by the dimensionality of these hidden layers. Each element of such a vector is simply a neuron, where each of these can be thought of as receiving many inputs and producing a single output. These neurons act in parallel to produce the layer's result to be fed to the next layer, and thus achieves a distributed representation of the input $\mathbf{x}$. [Goodfellow et al., 2016]

2.5.1 Example: The XOR Function

In section 2.4 we demonstrated how a linear model was unable to approximate the XOR function. This section will establish how a simple neural network can overcome the limitations of a linear model. We have four training examples that must be classified correctly, and they are given by

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},$$

where each row gives a single training example (one input vector, $\mathbf{x}$), and each column defines each element in the input vectors, (x_1, x_2), respectively.

We will use a neural network with one hidden layer. Further, our output function will be a linear function, as normalization will not be necessary for a binary result. Given the inputs, the hidden layer's result is computed with $\mathbf{h} = f^1(\mathbf{x}; \mathbf{W}, \mathbf{b})$, which is fed to the final layer giving $y = f^2(\mathbf{h}; \mathbf{w}, c)$, where $\mathbf{b}$ and c are separate bias terms. We then have

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}, \mathbf{w}, c) = f^2\left(f^1\left(\mathbf{x}\right)\right).$$

This relationship is illustrated in fig. 2.6, where the bias terms $\mathbf{b}$ and c will be connected to $\mathbf{h}^\top = (h_1, h_2)$ and y, respectively. The input and hidden layer are connected by the matrix $\mathbf{W}$, and the hidden and output layers are connected with the vector $\mathbf{w}$. [Goodfellow et al., 2016]

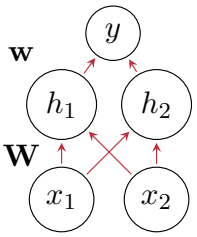

Figure 2.6: Illustrating the simple three layered neural network to model XOR function.

Note, in this example we have separated the bias terms from the parameter to keep it simple. If the bias terms were combined to the parameters the input vectors should be extended to match the dimensions.

2.5.1.1 Adding Nonlinearity

Nesting linear functions, such as $f^3\left(f^2\left(f^1\left(x\right)\right)\right)$, where all f^i are linear functions, can be rewritten as a single linear combination. And as described in the previous section, we know that a linear model is not able to approximate the XOR function. Thus, at least one hidden layer must perform a nonlinear transformation of the data representation. More specifically, the layer's neuron's activation functions must be nonlinear. Many such functions exist, but we will proceed with one called the rectified linear unit (see section 2.4.2), which is defined as

$$g(z) = \max(0, z). \tag{2.23}$$

The computation in the hidden layer will define an affine transformation from a vector $\mathbf{x}$ to a new vector $\mathbf{h}$, so each hidden unit will be computed as

$$h_i = g(\mathbf{W}_i\mathbf{x} + \mathbf{b}_i),$$

where $\mathbf{x}$ is a column vector, $\mathbf{W}_i$ gives the weights in the i^{th} row, and $\mathbf{b}_i$ is the i^{th} bias value. This gives us a complete neural network as [Goodfellow et al., 2016]

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}, \mathbf{w}, c) = \max\left(0, \mathbf{W}\mathbf{x} + \mathbf{b}\right)\mathbf{w} + c$$

and we assume that we have already learned the parameters (weights and bias terms) of the network, and they are given by

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \ \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \ c = 0.$$

2.5.1.2 Computing the XOR Function

Given the parameters we will calculate the output for the input data. First the neural network will multiply the input matrix by the first layer's weight matrix, $\mathbf{W}$, and add the bias vector $\mathbf{b}$ to that result. The activation function is then applied element-wise, which yields the result of the hidden layer. The final layer's weights $\mathbf{w}$ are then multiplied. Further, as c is zero, it does not influence the result and is simply omitted from the

calculation. Thus, we have

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \tag{2.24}$$

$$\mathbf{XW} + \mathbf{b} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \tag{2.25}$$

$$g(\mathbf{XW} + \mathbf{b})\mathbf{w} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \tag{2.26}$$

which is the desired result. If exactly one of the inputs are 1, the result is 1. Otherwise it is 0. Notice, that the final layer is still just a linear regression model as described in section 2.2, but the input is no longer the original $\mathbf{x}$, but the transformed $\mathbf{h}$.

2.6 Learning Representations

Responsible: Alexander Johansen (s145706)

As described in the previous sections, neural networks learn by attempting to minimise the given loss function with some gradient-based optimiser after performing predictions for the given input. The model receives some training input and performs some computations and gives a prediction which is penalised in accordance with what would have been the desired output.

Given the model's parameters (weights and bias terms) the loss function dictates how these should be updated by computing the partial derivatives for all the parameters to move towards a lower penalty. The approach is often the same as applied in updating linear regression (section 2.2) when computing continuous outputs and logistic regression (section 2.3) when handling categorical output. However, as a neural network is composed of multiple connected functions the computed partial derivatives (gradients) must be propagated back throughout each layer in the network. The formula used is known as Backpropagation (presented in section 2.6.1) which uses the chain rule to propagate all updates and find the gradient with respect to all weights and biases in the network. The gradients are then used with an optimisation algorithm (presented in section 2.6.2) to update the parameters. [Rumelhart et al., 1988, Goodfellow et al., 2016, Nielsen, 2015]

2.6.1 Backpropagation Algorithm

The backward propagation of errors (backpropagation) algorithm is often used for training neural networks with some optimisation method, e.g. gradient descent. The back-

propagation algorithm updates the parameters of a given network in the aim of minimising the desired loss function. This is done by computing the partial derivatives of the loss function w.r.t. any parameter in the network. The chain rule is applied to compute the gradients for each layer in the network. [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]

Let us define a neuron's activation function with input z as $\varphi(z)$ (see section 2.4.2), then the j^{th} neuron's output o_j is defined as

$$o_j = \varphi(z_j) \tag{2.27}$$

$$= \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right), \tag{2.28}$$

where z_j is the weighted sum of outputs o_k of the neurons in the previous layer, n defines the number of neurons which give input to the j^{th} neuron, and thus w_{kj} defines the parameter connecting the k^{th} and j^{th} neurons. Naturally, the input layer does not have previous layers, and the input is simply the current training sample.

The chain rule must be applied to compute the partial derivative of the loss L w.r.t. some parameter w_{ij} given the weighted sum z_j as

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}, \tag{2.29}$$

because the loss L is defined by output o_j, which is defined by z_j, which in turn is defined by w_{ij}. [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]

The last term. For the right-most term, $\partial z_j / \partial w_{ij}$, only one term in z_j depends on w_{ij}, which gives us

$$\frac{\partial z_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{k=1}^{n} w_{kj} o_k\right) \tag{2.30}$$

$$= o_i. \tag{2.31}$$

If i refers to the input layer then $o_i = x_i$. [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]

The middle term. The derivative of the j^{th} activation o_j w.r.t. its input z_j, $\partial o_j / \partial z_j$, is defined as the derivative of the activation function, $\varphi(z_j)$, as

$$\frac{\partial o_j}{\partial z_j} = \frac{\partial}{\partial z_j} \varphi\left(z_j\right) \tag{2.32}$$

$$= \frac{d}{dz_j} \varphi(z_j). \tag{2.33}$$

Thus, backpropagation is only applicable if the activation function is differentiable. [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]

The first term. Two outcomes are possible for the first term. The first possibility is that the output o_j can be the output of the output layer, and thus it is the network's prediction, i.e. $o_j = y$. The second possibility is that o_j is given from a neuron in a hidden layer.

If the first possibility is the case, then the first term is simply the partial derivative of the loss w.r.t. the prediction of the network. On the other hand, if the neuron is a hidden one, the loss must be computed as a combination of the loss for the neurons that receive input from the j$^{\text{th}}$ neuron, i.e.

$$\frac{\partial L(o_j)}{\partial o_j} = \frac{\partial L(z_1, \ldots, z_g)}{\partial o_j} \tag{2.34}$$

$$= \sum_{l=1}^{g} \left(\frac{\partial L}{\partial z_l} \frac{\partial z_l}{\partial o_j} \right) \tag{2.35}$$

$$= \sum_{l=1}^{g} \left(\frac{\partial L}{\partial o_l} \frac{\partial o_l}{\partial z_l} w_{jl} \right), \tag{2.36}$$

thus g defines the neurons receiving input from the j$^{\text{th}}$ neuron. This means that the hidden neuron's loss must be computed from the next layer's loss which in turn must be define beforehand. Note, that it must be possible to rewrite the given loss function as a function of a single training sample, otherwise backpropagation is not applicable [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006].

The first and middle term are often referred to as the error term δ_j of the j$^{\text{th}}$ neuron. Thus, the loss' derivative can be defined as

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} \tag{2.37}$$

$$= \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial z_j} o_i \tag{2.38}$$

$$= \delta_j o_i, \tag{2.39}$$

where δ_j is defined as

$$\delta_j = \frac{\partial L}{\partial o_i} \frac{\partial o_j}{\partial z_j} \tag{2.40}$$

$$= \begin{cases} \frac{\partial}{\partial o_j} L(o_j) \frac{d}{dz_j} \varphi(z_j) & \text{if } j \text{ is an output neuron} \\ \left(\sum_{k=1}^{g} w_{jk} \delta_k \right) \frac{d}{dz_j} \varphi(z_j) & \text{if } j \text{ is a hidden neuron} \end{cases} \tag{2.41}$$

where g again defines the neurons receiving input from the j$^{\text{th}}$ hidden neuron [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]. We do not specify neither the loss function nor the activation function here, but give a more general description of the error term.

Finally, the network's parameters are updated with some gradient-based optimisation, e.g. with gradient descent. We present some relevant optimisation algorithm in section 2.6.2. [Nielsen, 2015, Rumelhart et al., 1988, Bishop, 2006]

2.6.2 Optimisation Algorithms

This section presents some relevant optimisation algorithms. We present variants of the gradient descent optimiser (sections 2.6.2.1 to 2.6.2.3), and the Adam optimiser (section 2.6.2.4).

2.6.2.1 Gradient Descent

Batch gradient descent, or simply gradient descent, aims to minimise the loss function $L(\theta)$ parametrised by a model's parameters θ. Note, that the loss function must be differentiable. It considers the gradients of the entire set of training samples to take a step downhill towards a minimum w.r.t. the loss function's slope. Gradient descent takes a step in the opposite direction of the loss function's gradient and is given by

$$\theta \leftarrow \theta - \alpha \frac{\partial}{\partial \theta} L(\theta), \tag{2.42}$$

where α refers to the learning rate which defines how big a step we take w.r.t. to the loss function's gradient $\frac{\partial}{\partial \theta} L(\theta)$. The learning rate should be chosen to have a suitable value. If it is chosen with a small value the convergence will be very slow. On the other hand, if it is large it may hinder the convergence altogether. This is because the optimiser may overshoot the minimum on every update.

One challenge with regular gradient descent is the size of the data set at hand. Gradient descent considers the entire data set to take a single step. Depending on the size of the data set taking one step is time consuming.

2.6.2.2 Stochastic Gradient Descent

Stochastic gradient descent aims to address the challenge of a slow parameter update with regular gradient descent. Instead of using the entire training set only a single training sample is used to update the model's parameters.

Thus, every prediction the model makes is used to update the parameters. This way the process of updating the parameters is much quicker instead. However, these frequent updates only reflect what is known from the given training sample's target value, and thus the parameters updates do not guarantee a fall in the loss function's result at every update, i.e. the loss function will fluctuate heavily because of high variance.

Momentum. Desirably, if the optimiser continuously takes steps in the same direction, it would make sense to build up momentum as a ball rolling down a hill. This technique is known as momentum and aims to accelerate the updates for gradient descent. This is accomplished by adding a fraction of the previous update vector to the current one

$$\mathbf{v}_t \leftarrow \mu \mathbf{v}_{t-1} + \alpha \frac{\partial}{\partial \theta} L(\theta) \tag{2.43}$$

$$\theta \leftarrow \theta - \mathbf{v}_t, \tag{2.44}$$

where μ is a fraction to be added to the update vectors $\mathbf{v}_t$ at time t.

2.6.2.3 Mini-batch Gradient Descent

To overcome the high variance from the stochastic gradient descent but still have a quick parameter update (w.r.t. regular gradient descent) gradients can be computed for mini-batches of training samples. A mini-batch often contains a set of training samples which together secure a more directed step towards a minimum of the loss function.

2.6.2.4 Adam: Adaptive Moment Estimation

Adaptive Moment Estimation (Adam) is a first-order gradient-based optimisation algorithm for stochastic loss functions. It adaptively estimates the moments for speeding up optimisation of the loss function. Learning rates are computed individually for different parameters from estimations of first and second moments of the gradients. [Kingma and Ba, 2014]

The gradients g_t of the loss function $L_t(\theta_{t-1})$ w.r.t. the model's parameters θ at time step t is computed as

$$\mathbf{g}_t \leftarrow \frac{\partial}{\partial\theta} L_t(\theta_{t-1}). \tag{2.45}$$

The gradients are used to compute an exponentially decaying average of past squared gradients v_t and an exponentially decaying average of past gradients m_t. These are estimated as

$$\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t \tag{2.46}$$

$$\mathbf{v}_t \leftarrow \beta_2 \mathbf{m}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, \tag{2.47}$$

where Kingma and Ba [2014] propose to initialise $\beta_1 = 0.9$ and $\beta_2 = 0.999$. These moment estimates are biased towards 0, because they are initialised to 0. The authors counteract this bias by performing a bias-correction with

$$\widehat{\mathbf{m}}_\mathbf{t} \leftarrow \frac{\mathbf{m}_t}{(1-\beta_1^t)} \tag{2.48}$$

$$\widehat{\mathbf{v}}_\mathbf{t} \leftarrow \frac{\mathbf{v}_t}{(1-\beta_2^t)}. \tag{2.49}$$

The parameters are thus updated as

$$\theta_{t+1} \leftarrow \theta_t - \frac{\alpha \widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t} + \epsilon}, \tag{2.50}$$

where ϵ is a term to avoid division by zero and is often a small value, e.g. $\epsilon = 1 \times 10^{-8}$. [Kingma and Ba, 2014]

2.6.3 Regularisation

The class of function that a given neural network can represent is quite diverse. This is what makes neural networks so powerful because they are able to capture subtle nuances

and patterns in the dataset. It also makes the models extra susceptible to capture peculiarities in individual training samples and other small bits of noise. This failure mode is commonly known as *overfitting*, and it results in models that after training perform significantly more poorly when making predictions for never before seen test samples. The underlying reason for this is that the test samples simply do not look enough like the training data becuse they are unlikely to contain the same noise as the model learned to recognize from the training set.

There are a variety of clever techniques for preventing overfitting which we will look at.

As mentioned above, overfitting is a consequence of using models that have too much variety in what functions they can express which makes them able to fit the noise.

Make the model simpler. A straightforward method of reducing the model's excess *variance* is to simply limit what the model is able to express by making the model smaller, e.g. by reducing the amount of layers or that the model has, which will also reduce the amount of parameters in the model.

Weight decay. Another way to do it is by penalizing the model for choosing large parameter values by adding a weight decay term to the loss function: [Moody et al., 1992]

$$\mathcal{L}' = \mathcal{L} + \frac{1}{2}\lambda \sum_i \theta_i^2. \tag{2.51}$$

Here we have used the L2 norm of the parameters, θ, as an example, but the L1 norm is also used sometimes, or both variants can be combined. A weight decay term effectively limits the model's ability to freely choose the parameter values that best minimize the original loss function by pushing towards parameter values closer to 0. The regularization parameter, λ, controls how aggressively the parameter values should be pushed towards 0.

More data. Other techniques for avoiding overfitting generally entails ways of adding more noise during the training so the model can become less sensitive to it, and don't try to generalise on it. The ideal way to do this is by simply getting, and training on, a larger dataset. However, acquiring data can be hard, expensive, or even impossible, and there are alternatives.

Data augmentation. For many types of data it is possible to augment the samples in different ways that preserve the properties that we wish the model to learn, while still making them different enough that the model won't learn the same noise and individual peculiarities. For image data these augmentations typically entail cropping, rotating, or flipping along various axes.

Adversarial samples. For input in continuous spaces it is also possible to find the small change (which is often imperceptible to a human) in a training sample that is most likely to get the model to make a wrong prediction. [Szegedy et al., 2013, Goodfellow et al., 2014]

Dropouts. Another way to add noise inside the model itself is by using dropouts, which randomly sets units' activations to 0 during training which forces the model to have redundant means of conveying the actual signal in the data through the layers.

2.7 Recurrent Neural Network

Responsible: Jonas Hansen (s142957)

Recurrent Neural Networks (RNNs) are specialised neural networks for processing sequential data. RNNs are dynamic by nature and a network's parameters are shared across time steps. A RNN is constructed as a computational graph with loops for the different time steps. By unfolding an RNN we get a network that is similar to a neural network, but the parameters for each layer are shared across these layers. The size of the unfolded graph is proportional to the sequence's length. A dynamical system can be expressed as

$$s^t = f(s^{t-1}; \theta), \tag{2.52}$$

where s^t is referred to as the system's state. The state is recursively defined as a function of its past self with some parameters. To convince ourselves that this is similar to a neural network, consider unfolding s^t with $t = 3$

$$s^3 = f(s^2; \theta) \tag{2.53}$$

$$= f(f(s^1; \theta). \tag{2.54}$$

As with regular neural networks, the state of an RNN is to be understood as a hidden unit of the network. Now, given some sequential input, x^t, at time t we can represent an unfolded recurrence with a function g^t as

$$h^t = g^t\left(x^t, x^{t-1}, \ldots, x^2, x^1\right) \tag{2.55}$$

$$= f(h^{t-1}, x^t; \theta). \tag{2.56}$$

Furthermore, typically an RNN has an output layer that reads the final state and makes some prediction. The final state, h^t, can be considered as a lossy representation of the information that is relevant for the given task, up to time t.

Regardless of the length of the given sequence the RNN will always have the same input size. That is because it operates on the transition between two states instead of a variable sequence size. In fig. 2.7 we present the recurrent and unrolled graph representations of an RNN. The recurrent graph is a concise representation of the network, where the unrolled graph shows an explicit computational graph and the flow of information.

2.7.1 Backpropagation Through Time

Training an RNN is similar to training a regular neural network as explained in section 2.5. The network's gradients must be computed and a general-purpose gradient-based training technique can then use these gradients to update the network's parameters. The gradients are computed with the Backpropagation algorithm (which was introduced in section 2.6.1). The main difference of using backpropagation on an RNN is that the states from the previous time steps must also be used for computing the gradients. This is why the algorithm for RNNs is called Backpropagation Through Time (BPTT). [Goodfellow et al., 2016]

Thus, the BPTT algorithm must have the previously computed states at its disposal. Theoretically, this is done by unfolding the computational graph to a desired length and applying the backpropagation algorithm on the unfolded graph. Practically, the unfolding is not strictly necessary, because the RNN could be executed in a loop, where the states could simply be stored for future use. The key difference of BPTT from the regular backpropagation is that the gradients on each time step are summed together. The summed gradients can thus be used to update the network's shared parameters.

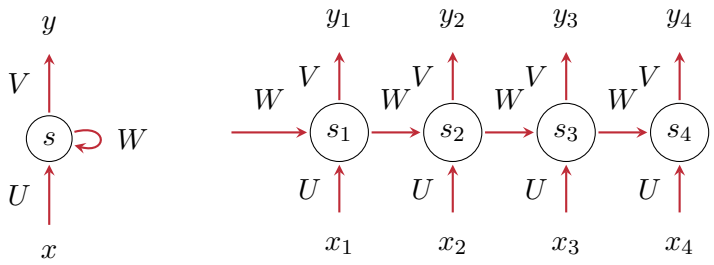

Figure 2.7: Illustrating an RNN with loop (left) and unfolding it (right).

Consider the rightmost graph in fig. 2.7. To compute the gradient for the loss L_i at time step i with the prediction y compared to the expected target t, we must sum the parameters of all the contributing states' gradients by using the chain rule, and it is similar to the following equation

$$\frac{\partial L_i}{\partial W} = \sum_{k=1}^{i} \frac{\partial L_i}{\partial y_i} \frac{\partial y_i}{\partial s_i} \frac{\partial s_i}{\partial s_k} \frac{\partial s_k}{\partial W}, \tag{2.57}$$

where y_i is the prediction at the state s_i and it is dependent on the previous i states. Thus, the gradient for every parameter in the RNN that is dependent on some previous state must be computed in a similar manner. The parameters that are independent of the previous states, intuitively do not incorporate the other states in computing the gradient. Thus, if we calculate the gradient for the parameters V, which are attached to the output layer, we get the following gradient computation

$$\frac{\partial L_i}{\partial V} = \frac{\partial L_i}{\partial y_i} \frac{\partial y_i}{\partial \left(V s_i\right)} \frac{\partial \left(V s_i\right)}{\partial V}. \tag{2.58}$$

2.7.2 Gated Recurrent Unit

Regular RNNs are great for sequential data. However, their performance decreases dramatically when long-term dependencies are needed to perform predictions. For instance if the context of a previous state is needed to predict the current state and that previous state is many time steps in the past a regular RNN would often not be able to capture such time dependency and fail to make the correct prediction.

Long Short-Term Memory (LSTM) were built to overcome the challenges of long-term dependencies of a regular RNN. A LSTM works by introducing gated computation in computing the next step. LSTMs have been shown to be capable of handling long-term dependencies much better than a regular RNN [Hochreiter and Schmidhuber, 1997]. This section presents The Gated Recurrent Unit (GRU) which is a simplified version of the LSTM. [Cho et al., 2014b, Goodfellow et al., 2016]

Gates of a GRU. Gated RNNs such as the GRU and LSTM use gates to control information flow with every time step. Gated RNNs use gates to better handle updating and forgetting their hidden representation of the current state of the neural network. In the GRU the gates regulate the amount of information to flow to the next hidden state with a sigmoid function (see section 2.4.2). The sigmoid's output determines if the information flow should stop entirely (0 as output), or let the information flow freely (1 as output). Using the sigmoid function the gate can control how the hidden state is to be updated and what to remember for the future hidden states. There are two gates in a GRU; the reset and update gates. The reset gate $\mathbf{r}_t$ and update gate $\mathbf{z}_t$ at time t are computed as

$$\mathbf{r}_t = \sigma\left(\mathbf{U}_r \mathbf{x}_t + \mathbf{W}_r \mathbf{h}_{t-1}\right) \tag{2.59}$$

$$\mathbf{z}_t = \sigma\left(\mathbf{U}_z \mathbf{x}_t + \mathbf{W}_z \mathbf{h}_{t-1}\right), \tag{2.60}$$

where σ is the logistic sigmoid function, $\mathbf{x}_t$ is the current input, and $\mathbf{h}_{t-1}$ is the previous state. $\mathbf{W}_r$, $\mathbf{W}_z$, $\mathbf{U}_r$, and $\mathbf{U}_z$ are learned parameters.

The activation of the unit, which is what is passed forward to the next core, is computed as

$$\mathbf{h}_t = (1 - z_t)\, \mathbf{h}'_t + z_t \mathbf{h}_{t-1} \tag{2.61}$$

$$\mathbf{h}'_t = \tanh\left(\mathbf{U}\mathbf{x}_t + \mathbf{W}\left(r\mathbf{h}_{t-1}\right)\right). \tag{2.62}$$

The reset gate allows the core to ignore any information that is not found to be relevant for future computation. The update gate is in charge of controlling how much information to use in the current state. Combined, these gates control the flow of information. [Cho et al., 2014b]

2.7.3 Vanishing and exploding gradients

Vanishing- and exploding gradients comes from the repeated application of the recurrent weight matrix (or saturating nonlinearities). This happens when the spectral radius of the recurrent weight matrix is either more than 1, which makes them explode, or smaller than 1, which makes the gradients vanish.

Vanishing gradients does not mean that all gradients vanish. Only some of them, gradients local in time will still be present. However, long-term dependancies will vanish, and the network be inept to learn thoese. [Hochreiter et al., 2001, Pascanu et al., 2012]

The GRU (section 2.7.2) can overcome the vanishing gradient problem, this is because of the gating mechanism where the forget gates gets to be the derivative. However, this does not overcome the exploding gradients problem, which we address by the following;

The most common way of dealing with exploding gradients is to either normalise the gradients or clip them. Clipping the gradients entail simply making sure that no gradient value is above a certain threshold.

Gradient normalisation simply means that the gradients are divided by their own L2 norm $\|\nabla_\theta\|_2$, and possibly multiplied by a constant, before the updates are applied. This way it is possible to ensure that the norm of the update is smaller than the chosen constant. The normalisation is normally only performed if the gradient norm of the gradient vector is greater than some predefined thresholds [Sønderby and Winther, 2014].

2.7.4 Encoder-Decoder Architecture

The RNN architecture, as described in section 2.7, can be modelled for many different types of sequential tasks. Figure 2.7 illustrated an RNN that produces an output $\mathbf{y}_t$ for every time step t. However, an RNN is often not dependent on producing an output for every time step. RNNs may be used for tasks requiring only one, some, or no outputs.

The encoder-decoder architecture is composed of two separate RNNs [Cho et al., 2014b, Sutskever et al., 2014, Goodfellow et al., 2016]. The encoder (input RNN) receives input from training samples, $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$. In [Cho et al., 2014c, Sutskever et al., 2014] only the final state $\mathbf{h}_{T_x}$ is transferred to the decoder, whereas in [Bahdanau et al., 2014] the entire sequence $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{T_x}$ is transferred. The hidden state(s) should be considered as a summarised representation of the original input.

The decoder (output RNN) receives the hidden states of the encoder and produces a sequence of predictions, $\mathbf{Y} = (\mathbf{y}_1, \ldots, \mathbf{y}_i)$. Note, the size of $\mathbf{X}$ and $\mathbf{Y}$ may differ. The output is based on the average probability of a sequence of predictions given a sequence of inputs. Thus, the encoder-decoder architecture is jointly trained to maximise the average $\log, p(\mathbf{y}_1, \ldots, \mathbf{y}_i \mid \mathbf{x}_1, \ldots, \mathbf{x}_k)$. [Cho et al., 2014b, Sutskever et al., 2014, Goodfellow et al., 2016]

The decoder uses the encoder by creating a *context vector*, $\mathbf{c}$, which is constructed by applying some nonlinear function, q, to the hidden states, which are produced from the

given input sequence $\mathbf{X}$, as follows

$$\mathbf{c} = q\left(\{\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}\}\right), \tag{2.63}$$

where $\mathbf{h}_{T_x}$ defines the last hidden state of the chosen RNN core (e.g. a GRU). The decoder is thus trained to give some next prediction $\mathbf{y}_{t'}$ w.r.t. all the previous predictions and $\mathbf{c}$. The combination of all these predictions is thus combined to a complete prediction $\mathbf{Y}$ with the joint probability of the conditionals

$$p(\mathbf{Y}) = \prod_{t=1}^{T_y} p\big(\mathbf{y}_t \mid \{\mathbf{y}_1, \ldots, \mathbf{y}_t\}, \mathbf{c}\big), \tag{2.64}$$

where T_y defines the length of the prediction. [Cho et al., 2014b, Sutskever et al., 2014, Bahdanau et al., 2014]

Attention Mechanism. The decoder is conditioned on the context vector. If its capacity is not large enough, the decoder will receive context that is very scarce and will thus have a hard time producing reliable predictions. Cho et al. [2014c] demonstrated this challenge by experimenting with longer input sentences, where the performance deteriorates dramatically with increasing input sizes.

An extension to the encoder-decoder architecture called the *attention mechanism* was proposed, where the idea is that the network learns to align the input data in a way such that the decoder can search for the relevant part of the encoder for the respective prediction. The decoder can then choose what context is relevant for the current prediction in combination with what it has already predicted. [Bahdanau et al., 2014]

Bahdanau et al. [2014] use a bidirectional RNN for the encoder. A bidirectional RNN consists of a forward and a backward RNN. The forward RNN reads the input in the given order and calculates a sequence of forward hidden states, $(\overrightarrow{h}_1, \ldots, \overrightarrow{h}_{T_x})$. The backward RNN reads the input in the reverse order and calculates a sequence of backward hidden states, $(\overleftarrow{h}_1, \ldots, \overleftarrow{h}_{T_x})$. They define an *annotation* as

$$\mathbf{h}_j = \left[\overrightarrow{\mathbf{h}}_j^\top, \overleftarrow{\mathbf{h}}_j^\top\right]^\top. \tag{2.65}$$

The decoder's conditional probability will now consist of the relevant context, $\mathbf{c}_i$, and not the entire context vector for every predicted word, $\mathbf{y}_i$, as

$$p(\mathbf{y}_i \mid \mathbf{y}_1, \ldots, \mathbf{y}_{i-1}, \mathbf{X}) = g(\mathbf{y}_{i-1}, \mathbf{s}_i, \mathbf{c}_i), \tag{2.66}$$

where g is a nonlinear function, and $\mathbf{s}_i$ is the decoder RNN's state at time i. $\mathbf{s}_i$ is computed w.r.t. to the previous state, previous prediction, and the relevant context vector as

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{y}_{i-1}, \mathbf{c}_i). \tag{2.67}$$

The context vector is computed as the weighted sum of the annotations, which the encoder provided, as

$$\mathbf{c}_i = \sum_{j=1}^{T_x} \alpha_{ij} \mathbf{h}_j, \tag{2.68}$$

where the weight parameter α_{ij} of each annotation $\mathbf{h}_j$ is computed as

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \tag{2.69}$$

and we have that

$$e_{ij} = a(\mathbf{s}_{i-1}, \mathbf{h}_j), \tag{2.70}$$

which is an alignment model, where α_{ij} and e_{ij} reflect the importance of $\mathbf{h}_j$, w.r.t. the previous decoder state $\mathbf{s}_{i-1}$, in deciding the next state $\mathbf{s}_i$. The alignment model is a neural network which learns to score how well the inputs around position j and the output at position i match. The alignment model is connected to the decoder and thus the gradients of the loss function can be propagated through, which means that it learns jointly with the rest of the network. [Bahdanau et al., 2014]

2.8 Embeddings

Responsible: Elias Obeid (s142952)

An embedding is a mapping from one space to another, such as $f : X \to Y$. Embeddings are thus very general, but often referred to within data science as being able to give alternative representations of data. [Goodfellow et al., 2016]

A popular usage of embeddings within data science is mapping discrete variables to dense vectors in a continuous space. The dimensionality of the embedding space can be chosen as a hyperparameter. This allows a distributed representation of discrete variables in which relationships with other discrete variables can be represented as distance between the variables.

While training a neural network the embedding matrix can be updated using the chosen optimisation algorithm to represent meaningful vectors for each discrete variable in order to perform that task [Bengio et al., 2006, Goodfellow et al., 2016]. Further, even though such embeddings might be of a high dimensionality it is popular to show the discrete variables' relationships with a 2/3D scatter plot using the t-Distributed Stochastic Neighbor Embedding (t-SNE) dimensionality reduction algorithm [Van der Maaten and Hinton, 2008].

The core idea of a distributed representation is to overcome the curse of dimensionality by learning a distributed representation of the data. The challenge is that when a model is being tested it will most likely encounter sequences of discrete variables it has not seen while training, i.e. it is hard to generalise well when trying to learn the joint distribution of discrete random variables. Any change to a discrete variable may have

a dramatic effect on the output of the estimated function. Thus, consider the amount of different sentences a model may observe, the number of values a discrete variable can take will result in most observed objects' hamming distance to be far from each other. For that reason it is important in high dimensions to distribute probability where it in fact matters and have some representation of similarity and relationship between for instance words. [Bengio et al., 2006, Goodfellow et al., 2016, Hinton, 1986]

Ideally, the embeddings would represent a meaningful relationship between for instance word pairs. Consider for instance the two sentences

> Jonas is writing a report.
> Alexander was reading an article.

Where each word is represented as a discrete variable. A human is able to form a meaningful relationship between words in sentences. In the example the sentences both start with a name, `is` and `was` are the same but in different tense (present and past tense), `writing` and `reading` are obviously related, and finally the object that is being written or read is properly conjugated/declined (`a` and `an` with `report` and `article`, respectively). For instance Mikolov et al. [2013a], Collobert et al. [2011b], Turian et al. [2010] have demonstrated that word embeddings are able to represent meaningful relationships between words, e.g. male/female, past/present tense, comparative/superlative, country/city, etc.

Mikolov et al. [2013b] showed that word embeddings could also be constructed from some context and thus sequences of words that have the same meaning should produce a similar word embedding. Such models are called "word2vec", because they encode words into vectors.

2.9 Translation Metrics

Responsible: Alexander Johansen (s145706)

This section will discuss some challenges and techniques that apply specifically when working with machine translation. We will present the perplexity measure for sequential data and discuss the use of Bilingual Evaluation Understudy to monitor the development of translation quality.

2.9.1 Measuring Perplexity

When using a sequence-to-sequence framework we can not simply apply the loss function given at section 2.3.2.1. Instead we must extend the cross entropy eq. (2.11) to include the sequential part.

We assume that $\mathbf{Y}$ is a matrix of predictions such that $\mathbf{Y}_{i,k}$ is a vector of predicted probabilities for the individual symbols. $\mathbf{T}$ is a corresponding matrix of targets such that $\mathbf{T}_{i,k}$ is a one-hot vector where all elements are zero, except the one that corresponds to

the true symbol for the k$^{\text{th}}$ element in the i$^{\text{th}}$ sample. Then the sequential cross entropy (sometimes called *perplexity*) is defined as

$$L(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{T_{t_i}}\sum_{k=1}^{T_{t_i}}\text{cost}(\mathbf{Y}_{i,k}, \mathbf{T}_{i,k}), \tag{2.71}$$

$$\text{cost}(\mathbf{y}, \mathbf{t}) = -\mathbf{t}^{\top}\log(\mathbf{y}), \tag{2.72}$$

where n is number of samples and T_{t_i} is the target sequence length of the i$^{\text{th}}$ sample.

2.9.2 BLEU: Bilingual Evaluation Understudy

The ultimate goal of building models for translation between languages is of course to help humans to understand and use information that is not available in a language they already master. In order to objectively follow our progress towards this goal, we need some quantitative way to measure the quality of the translations that our models produce. That means that given a translation, or a set of translations, we desire a single easily computable scalar number that correlates well with a typical human user's subjective opinion of the translation quality.

Judging the quality of a translation is a very subjective thing, and even professional human translators can produce very different translations. While it is sometimes most appropriate to translate a sentence literally word for word, in other cases it may be better to make a translation where the words mean something completely different when taken literally, but better carry the meaning of the original sentence while feeling more natural to the reader. Thus a single sentence in a source language may have a very large number of acceptable translations in another language that are potentially very different from each other.

In the field of translation (specifically machine translation) the BLEU[2] score [Papineni et al., 2002b] is a quite successful and widely used measure of translation quality that seems to be a somewhat fair indicator of translation quality as perceived by a human.

Unfortunately, even though BLEU is the primary measure that people care about when building machine translation systems, is not a continuous function and thus cannot be used for optimisation with gradient based functions. However, it is still very useful for assessing how good a given model is at making translations and for comparing the results.

In order to compute a BLEU score we need a hypothesis sentence, which in our case is a translation made by one of our models. Additionally we need one or more reference translations.

2.9.2.1 The modified n-gram precision

The most basic mechanism for computing a BLEU score is computing the fraction of n-grams that a hypothesis sentence has in common with it's reference sentences.

[2] Bilingual evaluation understudy.

In order to compute the *modified n-gram precision*, $p_{n,h}$, for a single sentence, h, we first need the following:

- Let $c_{n,h}$ be the number of n-grams in the hypothesis sentence h that also appear in one of the corresponding reference sentences. If some n-gram appears multiple times in h, it is at most counted as many times as it appears in the reference sentence where that n-gram appears the most times.
- Let $t_{n,h}$ be the number of n-grams in the hypothesis sentence.

Then the modified n-gram precision, $p_{n,h}$ for the hypothesis sentence, h, is

$$p_{n,h} = \frac{c_{n,h}}{t_{n,h}}. \tag{2.73}$$

Example[3] The highest number of times the uni-gram "the" appears in one of these reference sentences is 2, so when computing the modified uni-gram precision for this hypothesis, only the first two appearances of "the" are counted. Thus the modified uni-gram precision for this hypothesis is 2/7 even though the "traditional" uni-gram precision would have been 7/7 because all words in the hypothesis appear in a reference sentence.

Hypothesis the the the the the the the.
Reference 1 The cat is on the mat.
Reference 2 There is a cat on the mat.

Note that BLEU scores that are computed using more reference sentences are likely to be higher because there are more reference sentences in which to search for each n-gram.

It is often possible to make multiple sentences that carry almost the same meaning while using different words. This is also apparent when professional translators translate the same sentence, as they often produce different results with the same meaning. In order to compensate for this variability, and because we are more interested in the general quality of translations over an entire corpus than the quality of a single translated sentence, we compute the modified n-gram precision over an entire corpus all at once.

Let S be a corpus of hypotheses and their corresponding references. We compute the modified precision score, $p_{n,S}$, in the following way:

$$p_{n,S} = \frac{\sum_{h \in S} c_{n,h}}{\sum_{h \in S} t_{n,h}} \tag{2.74}$$

[3] from [Papineni et al., 2002b] example 2

2.9.2.2 Brevity penalty and putting it together

The final BLEU score is computed using the following formula by combining the modified n-gram precisions and multiplying by a *brevity penalty* (BP) factor:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log(p_n)\right). \tag{2.75}$$

Usually, only n-grams grams up to $N = 4$ are used, and they are weighted equally:

$$w_n = \begin{cases} \frac{1}{N} & \text{for } n \in \{1, 2, \ldots, N\} \\ 0 & \text{otherwise.} \end{cases} \tag{2.76}$$

The brevity penalty factor prevents giving good scores to very short hypothesis sentences that contain only very few n-grams, (almost) all of which appear in the corresponding reference sentences. Let h be the sum of all hypothesis lengths in the corpus, and let r be the sum of the lengths of reference sentences that are closest to having the same length as their corresponding hypotheses. Then the brevity penalty is computed by

$$\text{BP} = \begin{cases} e^{1-r/c} & \text{if } r \geq h \\ 1 & \text{otherwise.} \end{cases}$$

CHAPTER 3

Considerations and Tools

This chapter presents some of our relevant thoughts and considerations. First, we describe how we worked as a group and what tools we used throughout the project to manage our work. Afterwards, we mention why numerical challenges may occur (section 3.2) and present the relevant tools we used for building, running, and training models (sections 3.3 to 3.8).

3.1 Thoughts and Discussions of our Work Progress

Responsible: Jonas Hansen (s142957)

In this section we present and discuss the different tools and processes we used throughout the project. We will consider the communication channels and methods used to stay in touch with each other. Moreover, we will discuss how we managed to synchronise the work of all three group members without running into any conflicts or misunderstandings. This report is typeset with LaTeX using latexmk[1] to automate the document generation.

Throughout the project we aimed to keep sending a continuous stream of information every week to our supervisor so they would always know how far we were with the project and what challenges we were facing. In the following, the week numbers are counted from the beginning of the project period (not from the beginning of the year). Besides the regular personal meetings these "newsletter mails" were also a way for us to document what we did at what time. We have summarised these mails in appendix C.

3.1.1 Communication

Good communication is critical for proper progress with the project. There are many communication channels that we have at our fingertips without creating any new accounts. We for instance have regular text messages and Facebook's Messenger application to send and receive short messages. These are all great at messaging, but we know from experience that the many messages will become a massive mess after a few weeks. Such communication channels are great for short messages to give an update, e.g. that someone is running late.

For our day-to-day communication we used Slack[2], which is very user friendly and it provides many plug-ins to connect with other applications that we use on a daily basis.

[1] `https://www.ctan.org/pkg/latexmk/?lang=en`
[2] `https://slack.com/`

We would create different channels for different purposes, e.g. a channel for the report to discuss anything relevant to the report. It also provides private messages among the users in the group. It has file-sharing and sophisticated search capabilities among anything uploaded/written on Slack.

Slack's integration possibility is very useful. We for instance integrated our GitHub repositories with the respective Slack channels to keep track of changes made throughout the project. We present an example in fig. 3.1 of how Slack notifies if commits have been pushed to a respective repository. This is very handy as we may click commit's unique hash and view the changes directly on GitHub.

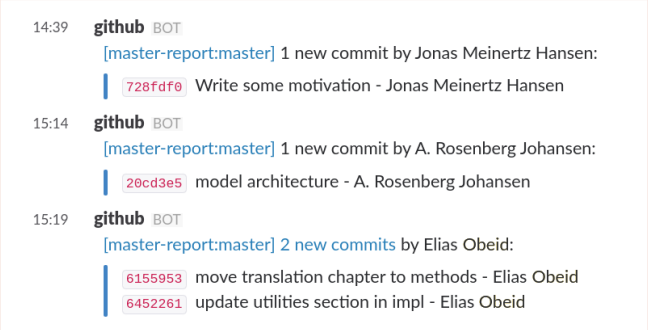

Figure 3.1: Example of integration of Slack with GitHub.

Moreover, we used Trello[3] as a board of tasks that should be completed. For every respective task we would document our discussions and ideas and make sure to move the tasks around to the correct columns to keep track our progress.

At one point we needed to start writing some report and to fuel our productivity we used BeeMinder[4] to track our progress towards a goal we defined personally. We aimed to reach 100 pages by the following week and with BeeMinder we could keep track of our progress and see if we were below the average to reach our goal. We never actually made our goal, but it was a great way to visualise how far we were.

3.1.2 Managing and Synchronising our Work

To keep track of our work we used git[5], which is an open source distributed version control system. We hosted our remote git repositories on GitHub[6] where each group member had access to the repositories.

We would create frequent local commits to keep track of our changes and push these to the remote repository for the rest of the group to see when we had working changes. With the ability to create branches with git we were able to change the code and experiment

[3] `https://trello.com/`
[4] `https://beeminder.com/`
[5] `https://git-scm.com/`
[6] `https://github.com`

without compromising the code on the master branch. Once we had tested and made sure our changes were correct we could merge the new branches into the master branch. Further, the integration with Slack made it extremely easy to keep everyone up to date with who did what.

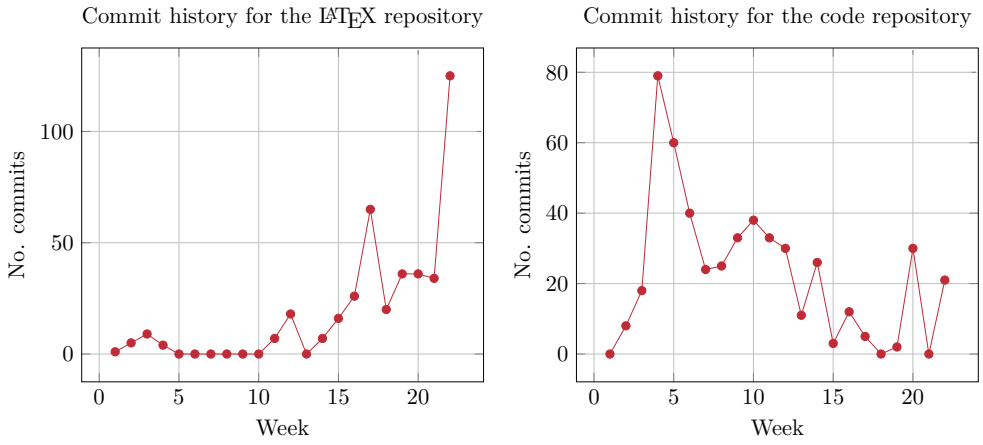

Figure 3.2: Curve over the development of our respective repositories.

We had two separate repositories for the source code and the report. Figure 3.2 illustrates two plots that visualise the number of commits for each week on the respective repositories. The figure nicely shows where our focus was for every week. We focussed mainly on developing our machine learning models in the first ten weeks. After ten weeks we slowly began working on the report and documenting our project. In the left figure the spike at week 17 represents the productivity we gained by tracking the development of the report with BeeMinder.

3.2 Numerical Challenges

Responsible: Elias Obeid (s142952)

Within the field of machine learning continuous variables are used to represent latent states and compute updates for optimizing functions. However, continuous variables pose a challenge for performing arithmetic operations on machines because these variables present an infinite space whereas machines have a finite amount of bits to represent values. Squeezing an infinite amount of numbers into a finite space can be done with an approximate representation, which is done by rounding the given value to fit in the finite space [Bryant and David Richard, 2010].

3.2.1 Overflow and Underflow

The approximate representation of numbers may lead to rounding errors, which can cause algorithms that work in theory to fail or yield incorrect results in practice [Goodfellow et al., 2016]. Two types of arithmetic rounding errors may occur; *underflow* and *overflow*.

If a floating-point operation results in a value smaller in magnitude than the smallest value representable, i.e. close to zero, then that value will be assigned to zero and one says that an underflow has occurred. On the other hand, if a floating-point operations results in a value larger in magnitude than the largest value representable (both positive and negative), then that value will overflow to some other value. Depending on the programming language, overflow can be presented by wrapping (modulus) the value, or by some other representation that indicates that an overflow has occurred, e.g. $\pm\infty$ or `NaN` (not a number). [Bryant and David Richard, 2010]

Underflow may cause mathematical operations such as logarithms and division to either fail or output an indication of a failure, e.g. $\pm\infty$, and further operations may result in `NaN` [Goodfellow et al., 2016].

As an example on rounding errors consider the cross entropy function in logistic regression (see section 2.3). Let $L(h_w(x), t)$ define the cross entropy of a prediction $h_w(x)$ and the expected value t, then we have

$$L(h_w(x), t) = -t \log(h_w(x)) - (1 - t) \log(1 - h_w(x)). \tag{3.1}$$

If $t = 1$ and $h_w(x)$ has underflowed then the function will attempt to compute the logarithm of zero, which is an undefined operation. Further arithmetic operations will result in unexpected results. To avoid unexpected results we apply clipping of $h_w(x)$:

$$\tau \leq h_w(x) \leq 1 - \tau, \tag{3.2}$$

where the tolerance, τ, is usually set to a value very close to zero, e.g. 1×10^{-10}, or some value given by $0 < \tau < \frac{1}{2}$ that guarantees that underflow does not occur.

3.3 Deep Learning Math

Responsible: Alexander Johansen (s145706)

While no individual operation, as presented in chapter 2, is in any way problematic to execute on its own, their collective computational cost quickly adds up when they are used together to implement a real model. We can illustrate the collective complexity with a simple dense layer in a neural network (NN). It requires a matrix multiplication, where the matrix size depends on the number of input neurons, output neurons and the batch size. Repeat this multiplication for each layer in the network. Add the operations for computing the gradients with Backpropagation while training. Each step taken with gradient-based optimiser quickly becomes expensive.

A recurrent neural network (RNN) is more complex and becomes even more computationally expensive to train than a typical “vanilla” neural network for two reasons:

1. Each time step in an RNN can be thought of as roughly analogous to a single layer in a “vanilla” neural network. However, the sequences that are used as input for RNNs can easily be much longer than even very large “vanilla” neural networks are deep, thus an RNN corresponds to a neural network with more layers.

2. More computations are needed to propagate information forwards (or backwards) through each time step in an RNN than for a simple dense layer in a "vanilla" neural network.

These circumstances along with the general tendency for deeper networks to perform better than shallow ones pushes researchers and other deep learning practitioners to crave for ever more computing power in order to to train deeper networks on larger datasets faster.

Amodei et al. [2015] mentions that training their DeepSpeech 2 model, which uses RNN for speech recognition, requires tens of exaFLOPs[7]. The performance of a single modern CPU is typically on the order of 100 gigaFLOPs[8]/second, so training such a network could take on the order of 10^8 to 10^9 seconds, or tens of years.

3.4 General-Purpose Graphics Processing Unit

Responsible: Jonas Hansen (s142957)

Unfortunately, in recent years Moore's law[9] has only kept true by adding more cores to the integrated circuits. This means that even though the total available computing power on a single integrated circuit has continued to increase roughly as observed more than 50 years ago by Moore, the increases in performance cannot be exploited by serial algorithms and writing parallel algorithms (if at all possible) is often difficult and imposes an overhead.

Fortunately, most of the mathematical operations required for training neural networks are relatively simple, and highly parallelisable, which allows for relatively easy speedups on modern hardware. Especially matrix multiplications can be quickly performed by a well optimised BLAS[10] library.

One class of hardware devices that are very capable of performing these simple highly parallel operations are Graphics Processing Units (GPUs). These devices are built for the purpose of calculating which color each pixel on a computer monitor should have. Modern monitors have several millions of pixels each, and it is not uncommon to connect multiple such monitors to the same computer, and every pixel should be updated around 60 times a second in order to ensure a pleasant experience for the user. In contrast with CPUs that are built with only single or double digits of cores that can perform instructions almost independently of each others, GPUs are typically built with hundreds or thousands of cores that are tied so closely together that they cannot carry out instructions independent of each other. This limitation is no problem when the goal is to update as many pixels as possible as fast as possible, since the computations needed

[7] 1 exaFLOP = 10^{18} Floating-point Operations.

[8] 1 gigaFLOP = 10^9 Floating-point Operations.

[9] Moore's law was formulated in 1965 by Gordon E. Moore, co-founder of Intel, and states that the number of transistors in an integrated circuit doubles approximately every two years. The observation has held roughly true for 50 years.

[10] Basic Linear Algebra Subprograms

for update each of the pixels are largely the same, only with small differences in input values.

With frameworks like Nvidia's CUDA it became possible to write more general instructions to be executed on the supported GPUs. This opened the possibility to use GPUs to train neural networks that are much faster than what was previously possible. Today Nvidia themselves acknowledges that deep learning is a significant and growing part of their business and supports the use by publishing libraries such as cuDNN, which provides highly optimized implementations of commonly used methods for Nvidia GPUs.

3.5 Deep Learning Frameworks

Responsible: Elias Obeid (s142952)

Over the years a number of frameworks have been developed for building and training neural network models. These frameworks help ease the workload by abstracting away some of the tedious details that need to be handled in order to run code on GPUs, such as copying data back and forth between GPU and main memory, making calls to the appropriate library functions, and computing gradients. The frameworks also often enable the user to define the models using a higher level language such as Python.

For the implementation of our own models, we decided to use the TensorFlow framework [Abadi et al., 2015], which was released as open source software by Google during the fall of 2015. The reasons for this decision was manifold, but some of the largest factors on the plus side include:

- Built-in support for distributing workload across multiple GPUs, or even across multiple different computers across a network
- It has an API for the Python programming language, which we are already familiar with
- Since the framework is backed by Google and had already managed to get some positive attention in the community we only expect that TensorFlow will become more popular and better supported as time goes by
- TensorFlow has a native tool called TensorBoard that serves both for visualising the computational graph, which makes up the model and for visualising tensor values and how they evolve during training (see section 3.6 for in-depth explanation)

TensorFlow belongs to a class of deep learning frameworks where the model is defined as a graph of individual tensor operations such as matrix multiplications. This way the frameworks are able to automatically do symbolic differentiation for the user with respect to the variables that they specify.

The main drawback of choosing TensorFlow is its relative immaturity compared to similar deep learning frameworks such as Theano [Theano Development Team, 2016], Caffe [Jia et al., 2014], Torch [Collobert et al., 2011a], and mxnet [Chen et al., 2015]. One aspect of this immaturity is that it is harder to find example implementations of specific network

architectures that we may want to use, simply because people haven't had the time to build them yet. It can also be harder to get help with specific problems because TensorFlow does not have so many users yet, and the users haven't had a lot of time to accumulate experience yet. Another aspect is that some features that exist in the competing frameworks have not been added to TensorFlow yet, and the features that are implemented are not very well tested in real world applications outside of Google.

3.6 TensorBoard

Responsible: Alexander Johansen (s145706)

In a sense neural networks are mostly black boxes, which perform computations based on a man-made computational graph. As mentioned in section 2.1 the hidden layers of a network are not dictated by the training data in any way. The many (millions) parameters in combination with nonlinear transformations makes debugging such a network very tedious and time-consuming. A network is trained in the aim of solving some predefined task to obtain a certain result.

Debugging the training process and the network itself is no easy task. TensorFlow includes a debugging visualisation system called TensorBoard, which can be used to monitor and debug the developed networks. It is used to create summaries of

- performance metrics during training
- keeping track of gradients
- visualising our computational graph
- tracking histograms for our variables with mean and several standard deviations

where the tools allow us to conveniently track and debug our model's performance while training. With TensorBoard it is much easier to keep track of the models and discuss if something might have gone wrong during training. This could for instance be done by viewing the summaries amongst the group and discuss what the graphs could mean. No one can be sure of the meaning, but we can make some qualified guesses and try to test our hypothesis by making the change and monitoring the summaries again.

As of this writing, TensorBoard consists of four tabs, events, graphs, histograms, and images. We elaborate on the former three tabs in the following sections, but first we give a few examples of how TensorBoard helped us debug our models. Note that figs. 3.3 to 3.6 are merely for illustrative purposes and not meant to give a visualisation of the actual contents of the figures.

3.6.1 Examples of Debugging with TensorBoard

In the following we give a few examples of cases where TensorBoard's visualisations helped us to reach a conclusion about whether or not our model needed modifications to work better.

Overfitting Rather Quickly. In the early stages of our project we were unable to get a decent translation performance on the data set. With the help of our supervisors we noticed that the model was overfitting the data set rather quickly. This was very peculiar and we were unsure of why it was overfitting so badly. It turned out that we had a severe mistake in our data loader, where it would only read the first 16 or so data samples (we had millions of data samples) on every iteration. Fixing this mistake showed that the model was no longer overfitting the data and the translation performance sky-rocketed.

Unstable Performance Metric. Throughout the project we have been interested in the translation performance of the developed models. At one point we noticed the BLEU validation score varied a lot after approximately 20,000 iterations. Our hypothesis was that the learning rate was too large for this point of training and the network was overshooting the minimum. Testing this assumption verified our hypothesis and we lowered the learning rate.

Increasing Bias. During training we noticed that the gradients were exploding, i.e. their values increase a lot. Exploding gradients may occur when activations functions may output large values, which in turn may cause a prediction to be way off. Making large updates to the parameters of the network may cause a model to perform badly because the different layers may be unstable after the large update. Vanishing gradients is the opposite problem, where the gradients become too small. To circumvent such challenges we installed gradient clipping and normalisation in our models.

3.6.2 Events

To keep track of events occurring while training, the TensorBoard events tap allows logging training and/or validation accuracies along with other metrics. We ran our models for 160,000 iterations and while training we monitored the:

- average character-by-character accuracies
- average character-by-character cross entropy loss
- global normalised gradients

where these three training summaries were written to the logs every 500^{th} iteration, which is a setting defined in the model. We show the events tab in fig. 3.3, which displays three graphs showing the development of their respective values.

Occasionally the model would halt the training process to perform a validation run with the model. We present the five validation summaries in fig. 3.4. These summaries monitor the following:

- average character-by-character accuracies
- average character-by-character cross entropy loss
- average edit distance per character

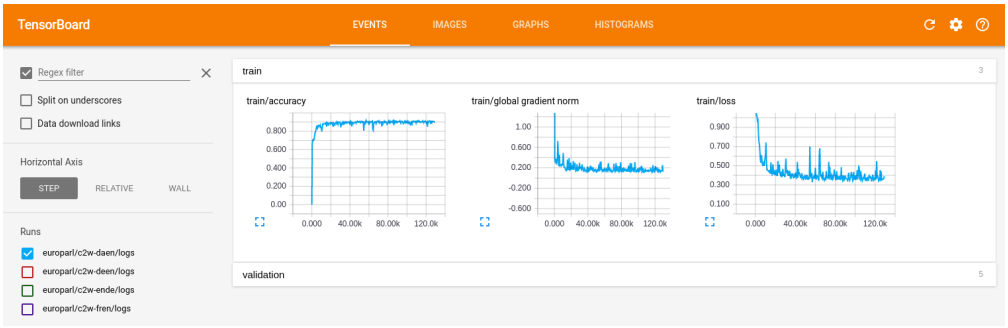

Figure 3.3: TensorBoard's events tab visualising the three training summaries.

- BLEU performance from moses-smt[11]
- our custom BLEU implementation (as presented in section 4.7.1)

where these summaries where saved every 3000$^{\text{th}}$ iteration.

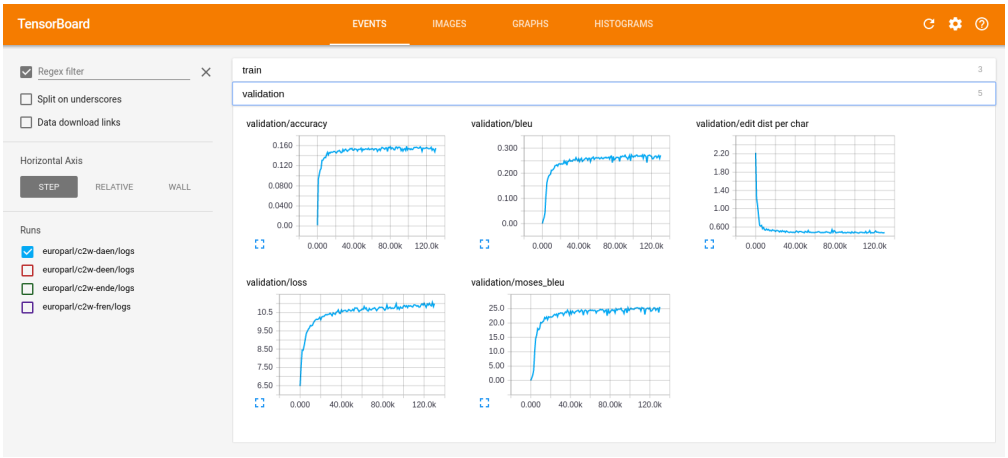

Figure 3.4: TensorBoard's events tab visualising the five validation summaries.

3.6.3 Graph

TensorBoard's graphs tab allows gives us an interactive visualisation of the computational graph and the flow of data of the model. This can be used to debug the implemented network to make sure that it is as expected. The graph is illustrated in fig. 3.5 and is organised according to TensorFlow's namespaces.

Interpreting the given graph can be quite challenging because such graphs may very quickly become very large and encompass a lot of different connected layers. TensorBoard

[11] `http://www.statmt.org/moses/` is a statistical machine translation system from which we used the `multi-bleu.perl` implementation to validate the BLEU performance.

Figure 3.5: TensorBoard's graph tab visualising the computational graph of our model.

gives a description of the different possible components in such a graph in the lower left corner of fig. 3.5. Moreover, if the developer has not defined different components of the graph in suitable namespaces it will be a massive mess to view. Further, it seems as TensorBoard has a hard time visualising our custom RNN layers based on TensorFlow's `while()` operation.

3.6.4 Histograms

In our opinion the naming of TensorBoard's tabs is not at first very intuitive. In the histogram tab we asked TensorFlow to monitor the parameters of the model. Compared to the events tab, which plots the data as curves, the histogram tab yields results as histograms. Histogram of one of our layers (the decoder with a GRU and attention) is visualised in fig. 3.6. The lighter color defines a higher standard deviation, and darker colors define lower standard deviation.

3.7 The University's GPU Servers

Responsible: Jonas Hansen (s142957)

The Technical University of Denmark (DTU) provides about half a dozen remotely accessible Unix servers each with two to four GPUs. These servers provide us with the necessary computational power we need to train our models within an acceptable time frame.

The servers are not directly accessible from the outside. They are however connected to the Internet, but one must connect to a password protected server (the thinlinc server) and from here connect to the respective GPU servers This thinlinc server is on the same local area network as the GPU servers and it is configured with connection information

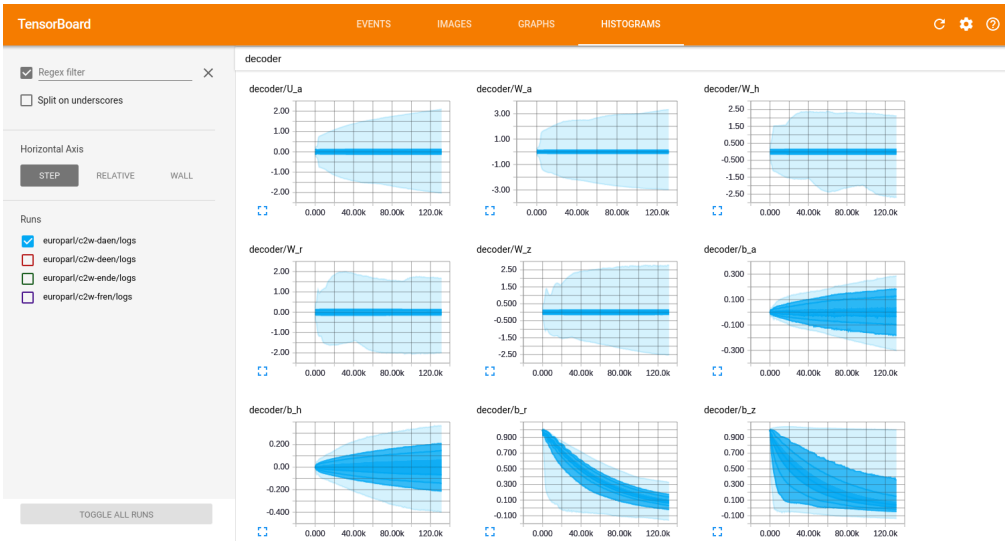

Figure 3.6: TensorBoard's histogram viewing the histogram page of the decoder

for each of them. We used encrypted ssh connections to the servers as illustrated in listing 3.1.

Listing 3.1: Connecting to the Theia GPU server via the thinlinc server.

```
1 ssh userA@thinlinc.compute.dtu.dk
2 # password prompt; connected to 'thinlinc.compute.dtu.dk' as 'userA'
3 ssh userB@theia
4 # password prompt; connected to 'theia' server as 'userB' via 'thinlinc server'
```

Note, that `userA` and `userB` are not necessarily different. The thinlinc server has preconfigured ssh connection information, and thus the `theia` server is one of the GPU servers. They are all named after Greek Titans.

3.7.1 Setting up the Servers for TensorFlow

The University's servers have been setup by an IT team and we are not allowed to make too many changes to the setups. The TensorFlow framework assumes some default locations for the GPU's `.so` files (shared object files) which is not in accordance with the setup of the servers. Therefore, we create symbolic links referring to the original `.so` files. We then provide our TensorFlow setup with the relevant paths to these (see section 3.8.2).

Listing 3.2 presents the bash script which creates these symbolic links. It only executes if the folder, which will hold the symbolic links, does not already exist. It then creates symbolic links in the root folder, creates the folder to hold them, and moves and renames

all created files. Note, that we only manipulate the symbolic links and not the original files on the system. This script is thus only run once on each machine.

Listing 3.2: (`setup/bashrc.symlinks`) Create symbolic links if they do not exist.

```
if [ ! -d $HOME/tensorflow-cuda-symlinks ]; then
  # create symlinks
  for f in /usr/local/cuda/lib64/*.so
  do ln -s $f ~/; done
  # create folder for symlinks
  mkdir ~/tensorflow-cuda-symlinks/
  # rename all symlinks
  mv ~/*.so ~/tensorflow-cuda-symlinks/
  for f in ~/tensorflow-cuda-symlinks/*.so
  do mv $f $f.7.0; done
  # rename libcudnn symlink
  mv ~/tensorflow-cuda-symlinks/libcudnn.so.7.0
   ↪  ~/tensorflow-cuda-symlinks/libcudnn.so.6.5
  cp ~/tensorflow-cuda-symlinks/libcudnn.so.6.5 ~/tensorflow-cuda-symlinks/libcudnn.so
fi
```

Moreover, TensorFlow assumes that the CUDA and cuDNN environments are of some specific version and naming. We used trial and error to figure out what the framework wanted the naming of the files to be and adjusted the symbolic links accordingly. These symbolic links are then used to configure the TensorFlow framework to use the available GPUs. In section 3.8 we present more details about how we used these symbolic links and other configuration with TensorFlow.

3.7.1.1 Serving Relevant Data to the Models

Downloading the required data for training and validation was also a challenge, because such files contain quite a bit of data. Even though it is rarely hard to find unused GPUs, it may not always be possible to use the same server, so we could not simply download the data we need on a single server and expect to always use that. In order to be able to quickly get the needed data on a new server whenever we needed it, we made a simple `bash` script to download the files (with the `curl` command) we needed from our personal Dropbox. We used Dropbox for hosting the data ourselves both because it gave us finer control over which data to download (compared to getting a compressed container with lots of large files that we don't actually need for our purposes), and we also found that the download speed from Dropbox was much faster compared to the original host on `statmt.org`.

3.7.2 Monitoring Servers for Targeted Access

The GPU servers are used by many for various tasks at the University. Thus, when we occasionally needed to train and monitor one or more of our models we would have to

sign in to every server to see if it is available. This was a tedious process, so we aimed to automate the process and make it easier for ourselves in the future.

We created pairs of asymmetric encryption keys and placed them on the respective GPU servers along with the matching key on the host server (the thinlinc server)[12] for passwordless access from and to authorised servers. This enabled us to write a script that combines GPU information for every server we desire. Listing 3.3 shows how we access the `theia` server and fetch its GPU information. As `theia` holds two GPUs is yields two lines representing relevant GPU information. With the presented script we fetch the fan's activity (percent), the GPU's temperature (Celsius), its current performance (`P0` is max and `P12` is min)[13], its power consumption (current/max), its memory usage (current/max), its overall usage (percent), and which type of compute capability the GPU has.

Listing 3.3: Bash script to fetch relevant GPU info remotely.

```
1 ssh theia nvidia-smi | grep '%' | sed 's/|/ /g'
2 # the theia server has two GPUs and the script yields the following
3 #  33%   72C    P0   139W / 235W    7269MiB / 11519MiB      88%      Default
4 #  65%   85C    P2   183W / 250W    4131MiB / 12287MiB      94%      Default
```

We are then able to sign in to the thinlinc server and run a script that connects to all GPU servers and prints their GPUs' status. From this we are able to efficiently target our model training towards idle servers. The entire script is presented in appendix B.1.

3.8 TensorFlow inside Docker Containers

Responsible: Elias Obeid (s142952)

In section 3.7 we mentioned half a dozen servers equipped with GPUs. Moreover, we use the TensorFlow machine learning framework to build and run our models. As we have mentioned, TensorFlow is a relatively young framework experiencing frequent updates and modifications. Throughout the project we aimed to stay up to date with the latest releases from the TensorFlow developer team and this means that we must update our server setup on quite a few occasions. For this we use Docker[14], which is a containerisation platform. With Docker we are able to build a reusable image and from that image create containers which run a virtual instance of the image. It is thus up to us to configure the image with what we need. With easy remote distribution of these images Docker is perfect for our needs.

We made a custom Dockerfile and hosted the built image on Docker Hub[15], which works

[12] `http://www.linuxproblem.org/art_9.html` gives a short guide on how to setup SSH login without passwords.

[13] See the `nvidia-smi` manual (`http://developer.download.nvidia.com/compute/cuda/6_0/rel/gdk/nvidia-smi.331.38.pdf`) under `Performance State`.

[14] `https://www.docker.com/`

[15] `https://hub.docker.com/r/obeyed/py3-tf-gpu/`

as a repository for Docker images to be accessible from anywhere. We present our Dockerfile in listing 3.4, which is based on Nvidia's official CUDA[16] image with the cuDNN[17] runtime. We are thus able to build upon the official CUDA Docker image and add our own custom changes, e.g. we install TensorFlow, Python, and other tools we need for training models.

Listing 3.4: (`setup/docker/Dockerfile`) Custom Dockerfile from which we build Docker images based on TensorFlow nightly builds.

```
1  FROM nvidia/cuda:7.5-cudnn4-runtime
2  MAINTAINER Elias Obeid <ekobeid@gmail.com>
3  # install dependencies and tools
4  RUN apt-get update && apt-get install -y python3-pip python3-dev libjpeg-dev libjpeg8-dev
   ↪  python3-matplotlib build-essential libblas-dev liblapack-dev libatlas-base-dev
   ↪  gfortran vim git curl python-skimage && apt-get clean && rm -rf /var/lib/apt/lists/*
5  # fetch and install nightly build for TensorFlow
6  RUN pip3 install --upgrade http://ci.tensorflow.org/view/Nightly/job⌋
   ↪  /nigntly-matrix-linux-gpu/TF_BUILD_CONTAINER_TYPE=GPU,TF_BUILD_IS_OPT=OPT,TF_BUILD⌋
   ↪  _IS_PIP=PIP,TF_BUILD_PYTHON_VERSION=PYTHON3,label=gpu-working/lastSuccessfulBuild⌋
   ↪  /artifact/pip_test/whl/tensorflow-0.8.0-cp34-cp34m-linux_x86_64.whl
7  # install python3 dependencies
8  RUN pip3 install numpy scipy matplotlib sklearn click nltk pandas
9  # initialise bash for container
10 CMD ["/bin/bash"]
```

3.8.1 Building and Updating Docker Images

We developed a small script which builds and takes care of tagging the newly built image according to our settings. This script is presented in listing 3.5. After building and updating the image and its version we push the image to Docker Hub for future use. This is done with a `git` like syntax with `docker push`. We have of course set up Docker to connect to the correct user and repository on Docker Hub beforehand.

Listing 3.5: (`setup/docker/docker-build-tag.sh`) Our build script which builds and updates the image's tags according to the version and image name given.

```
1  VERSION=x.y.z
2  IMAGE=obeyed/py3-tf-gpu
3  ID=$(docker build  -t ${IMAGE} . | tail -1 | sed 's/.*Successfully built \(.*\)$/\1/')
4  # tag docker image with new version and set it to latest
5  docker tag ${ID} ${IMAGE}:${VERSION}
6  docker tag ${ID} ${IMAGE}:latest
```

[16] https://developer.nvidia.com/cuda-zone
[17] https://developer.nvidia.com/cudnn

The idea is thus that the version number defined in listing 3.5 (`VERSION`) is to be used in a global Docker version variable (`GPU_DOCKER_IMAGE_VERSION`). This global variable dictates if the current system has the newest image version present on the machine or it must be fetched from Docker Hub. We present the Docker setup in detail in the following section.

3.8.2 Creating and Running Containers

When we need to create a Docker container from our image we need to provide the relevant drivers which TensorFlow needs to be able to run on the GPUs. These are the shared object files mentioned in section 3.7.1. With Docker it is straightforward to start a new container from an image with `docker run -it IMAGE:VERSION`, but with TensorFlows dependencies and the setup needed we have constructed aliases that take care of starting Docker containers on the servers.

Listing 3.6: (`setup/bashrc.envvars`) Script that exports new environment variables to system.

```
1 export CUDA_SO=$(\ls /usr/lib/x86_64-linux-gnu/libcuda* | xargs -I{} echo '-v {}:{}')
2 export DEVICES=$(\ls /dev/nvidia* | xargs -I{} echo '--device {}:{}')
3 export LD_LIBRARY=$(\ls $HOME/tensorflow-cuda-symlinks/* | xargs -I{} echo '-v {}:{}')
4 export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64/:$HOME
  ↪ /tensorflow-cuda-symlinks/"
5 export CUDA_HOME=/usr/local/cuda
6 export GPU_DOCKER_IMAGE_VERSION=x.y.z
```

The purpose with the exported environment variables presented in listing 3.6 is to keep a structured overview of what is fed to the Docker containers. These are used by the TensorFlow framework to work with a GPU. The `GPU_DOCKER_IMAGE_VERSION` is manually updated every time we create a new image. The `CUDA_SO` defines the original location of the CUDA libraries. The machines GPU devices are listed in the `DEVICES` variable. The `LD_LIBRARY_PATH` defines i.a. where our symbolic links (with updated names) are located. These are thus fed to our alias presented in listing 3.7.

Listing 3.7: (`setup/bashrc.aliases`) Script that creates aliases for new docker commands to run containers correctly on system.

```
1 alias gpu-docker='docker run -v $HOME:$HOME -w="$HOME" -e
  ↪ LD_LIBRARY_PATH=$LD_LIBRARY_PATH -e HOME=$HOME -e LC_ALL=C.UTF-8 -e LANG=C.UTF-8 -it
  ↪ $LD_LIBRARY $CUDA_SO $DEVICES obeyed/py3-tf-gpu:$GPU_DOCKER_IMAGE_VERSION'
2 alias gpu-docker-port='docker run -p 6007:6006 -v $HOME:$HOME -w="$HOME" -e
  ↪ LD_LIBRARY_PATH=$LD_LIBRARY_PATH -e HOME=$HOME -e LC_ALL=C.UTF-8 -e LANG=C.UTF-8 -it
  ↪ $LD_LIBRARY $CUDA_SO $DEVICES obeyed/py3-tf-gpu:$GPU_DOCKER_IMAGE_VERSION'
```

The `gpu-docker` alias is used to start a container with all dependencies loaded and ready

for TensorFlow model training. We added the `gpu-docker-port` because we for some training sessions want to monitor the development of the training for a model as it is training. This is done with TensorBoard and we must open a port through which it can connect to TensorFlow and receive training data. We did not include the port in the first alias because only one process can have the same port open. Thus, we would not be able to run more than one model on a server if they all open a port.

Listing 3.8: This piece of code makes sure to load our dependencies and settings from the setup folder.

```
1 if [ -f $HOME/folder/setup/bashrc.symlinks ]; then
2   . $HOME/folder/setup/bashrc.symlinks
3   . $HOME/folder/setup/bashrc.envvars
4   . $HOME/folder/setup/bashrc.aliases
5 fi
```

Now, instead of copying all dependencies and setting up every server, we put together a small piece of code to be included in the `.bashrc` of each server (see listing 3.8). Every time we signed in to a new server (that was not configured yet) we add the piece of code in listing 3.8 to the `.bashrc` file and sign out and in again for the changes to take affect. Notice that the `folder` in the path must correspond to the correct path for the repository. The code first checks to see if the first file that it needs exists in the respective folder, and then it updates the system with the new dependencies. Now the `gpu-docker` alias would be usable as it is defined in the `bashrc.aliases` file.

We used git with GitHub to manage and distribute our source code. Every time we made an update to either the source code or the dependencies defined in the `setup` folder we would simply pull the changes from GitHub. If any changes were made to the setup files we would have to sign out and in to the server for the changes to take affect.

CHAPTER 4

Implementation

This chapter presents and describes our implementation efforts and any relevant thoughts and choices made throughout the project. The implemented components and their dependencies are presented in fig. 4.1. The figure is meant as an illustrative overview of the connectedness of the various components that constitute the entire training process.

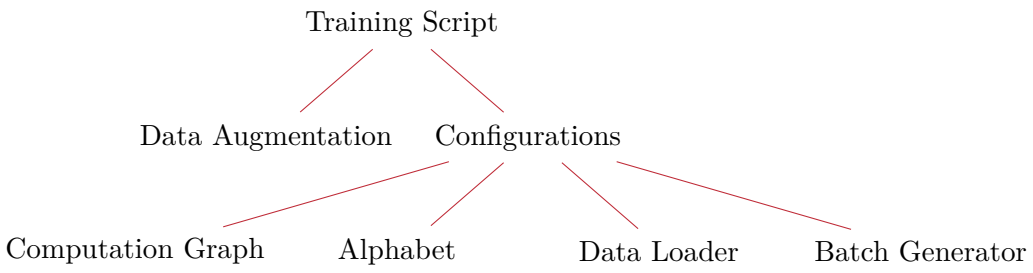

Figure 4.1: Overview of relevant components and their connectedness.

The training process begins with the training script which needs a configuration to build a trainable model. The model is defined as a computation graph which dictates how data is propagated through the network to perform predictions and also how the network is updated to minimising the prediction error. The training samples are loaded from disk by the data loader from which the batch generator serves formatted batches to the model and training step. The alphabet is constructed before training and it defines the symbols that the model will be able to understand.

The data in each batch may be subject to some form of augmentation to add noise to the data. This augmentation is performed w.r.t. to a schedule defined in the configuration. The training script instantiates the data augmentation process and proceeds according to the schedule.

The following sections will i.a. describe the above-mentioned components in greater detail. We start with section 4.1 giving a more formal definition of the task of translating sentences. Section 4.2 gives an overview of our data loader framework, while the more specific details are given in section 4.3 about the alphabet of symbols and section 4.4 about the batch generator and the iteration schedules.

The setup for writing model configurations is described in section 4.5 and the training script that ties all the other parts together is described in section 4.6. Finally, section 4.7 walks through some useful utility functions that we implemented and section 4.8 gives an

overview of some customized TensorFlow functionality that we have implemented along the way.

4.1 The Formal Task at Hand

Responsible: Alexander Johansen (s145706)

In section 1.4 we informally introduced the task at hand, where we now wish to define it more formally. To infer a mathematical description of our task, we must define how we model the space of a symbol. A symbol, s, given from a finite set of symbols, S, of size m is modelled using one-hot encoded vectors, i.e. $s \in \{0,1\}^m$, where $\sum(s) = 1$. The purpose of this encoding is to have each row in the vector represent a symbol in S. The index at which a 1 is present defines the given symbol, and the index may be used as an identifier for the symbol. A sequence of symbols, x, of length T_x is thus modelled as $x \in \{0,1\}^{T_x \times m}$, where $\sum(x) = T_x$. The task of converting language pairs can be defined as a mapping given by

$$f : \{0,1\}^{T_x \times m} \to \{0,1\}^{T'_x \times m'}. \tag{4.1}$$

Notice that the source and target sequences may differ in length and in the size of finite symbol space [Cho et al., 2014c]. Furthermore, as the vector of symbols has a tendency to grow in size we propose a trick in section 2.8 to transform the input sequence

$$f : \{0,1\}^{T_x \times m} \to \mathbb{R}^{T_x \times e}, \tag{4.2}$$

where e is a configurable parameter and is often $e \ll m$. We define a dataset, X, consisting of data points of sequences of symbols, x, as a collection of n data points, such that a dataset can be denoted as $X \in \mathbb{R}^{n \times T_x \times e}$.

Figure 4.2 illustrates how a sequence of symbols is modelled into individual character embeddings. The sequence of symbols (fig. 4.2(b)) consists of the consecutive symbols:

`This is a test`,

with an end-of-sequence symbol append, ω, and it has been padded with two symbols. Input sequences in a minibatch are padded to have the same length. The input mask (fig. 4.2(a)) defines which of the symbols are relevant for the input sequence. It thus informs that the final two symbols are to be ignored, because they are not part of the original sequence. The symbols' identifiers (fig. 4.2(c)), which are given by the one-hot encoded vectors, are then used to extract the symbol's respective embedding (fig. 4.2(d)), which is referred to as a character embedding. We illustrate the embeddings as column vectors with color coding, where a darker color defines a higher value at the vector's position.

4.2 The Data Loader

Responsible: Jonas Hansen (s142957)

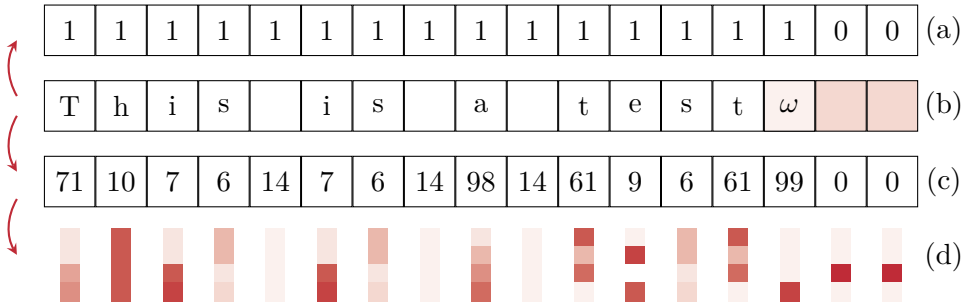

Figure 4.2: Example of how an input string is encoded to character embeddings.

These sequences of symbols are given by various datasets. The datasets define the source and target sequence, i.e. the input and expected output. We present the datasets relevant for this project in section 5.1.

Loading data and preparing it to be fed into the model is an important and very hard part of building a system for training neural networks. Especially combining different datasets can be particularly challenging because each dataset may be provided in a different formats that all requires the user to learn the specifics before being able to include them in the system.

To handle the loading, scheduling, and processing of data that is needed during training, we made a fairly flexible text data loader, which is mainly located in the file `text_loader.py`. The `TextLoader` class has two relevant methods, `_load_data()` and `_preprocess_data()`. An instance of the class is invoked with the relevant paths for the source and target data along with the wanted sequence length. The text data loader is used directly by the configuration files (see section 4.5) as needed, and can be thought of as performing three overall tasks:

1. Loading and preprocessing the text (text data loader)
2. Sample/minibatch scheduling (iteration schedule)
3. Formatting data for the model (batch generator)

The first task, loading and preprocessing the text, is done simply by reading the text files that contain the source and target sentences respectively and storing them in memory in a way that allows us to keep track of which source and target sentences that correspond to each other. It is at this point that we do some basic filtering of the data by discarding sentences that are too short (less than a few characters) or truncating sentences that are too long (the cutoff threshold is defined from the configuration).

In addition to regular bucketing of samples according to the sequence lengths, we have also found that employing some deliberate strategies for determining the order that data is trained on can help speed up the training. Part of the data loader is a sophisticated set of functions for intelligently controlling the order in which the data samples are yielded

by the text data loader for the model. This mechanism, given by the iteration schedule, is described in detail in section 4.4.1.

The last main function of the text data loader is to take a list of strings that has been marked (by the iteration schedule) for inclusion in the next minibatch, and convert them into a format that is appropriate as inputs for the computation graph that represents the model. This is carried out by the batch generator, which combines training samples into minibatches to be fed to the model. We describe the batch generating process in detail in section 4.4.

4.3 The Alphabet

Responsible: Elias Obeid (s142952)

As mentioned above the available datasets may be formatted and configured differently, and may contain many different symbols. We construct an alphabet which defines all valid symbols the model will be able to use. If a symbol is not present a special symbol is added to indicate that the symbol encountered on this position was unknown. The `Alphabet` class has two relevant methods, `encode()` and `decode()`.

The class is instantiated with either a list defining the wanted alphabet of symbols or a string to which a previously constructed alphabet file has been saved. We have constructed a list containing all symbols from the datasets where each symbol has a number of occurrence in the respective dataset. This list is pickled and saved as a binary file to be loaded and used for every run. The idea is that we can thus load the alphabet file and define the minimum number of occurrences a symbol must have before it is to be used in the list of available symbols for the training run.

The alphabet is precomputed by running through each of the data files and counting the number of times that each character occurs. Afterwards the list is cut off at a threshold ϕ which can be used in one of two ways. We either make the alphabet such that it contains characters that appear at least ϕ times each in the dataset, or we make the alphabet such that it consists of the ϕ characters that appear most often in the dataset. It is very common to use the second method with $\phi = 300$ so the alphabet consists of 300 known characters.

The `encode()` method translates a string to a sequence of integer values w.r.t. the list of symbols the class instance was invoked with. The `decode()` method translates the encoded integer values back to readable strings, and it replaces any unknown symbols with a special symbol indicating that the symbol was not in the constructed alphabet. Thus, the encoding step traverses a sequence of symbols and translates these into the respective symbol's index from the alphabet list. This step is illustrated in fig. 4.2 where the input sample (part (b) of the figure) is translated into identifiers (part (c) of the figure). The process may be reversed by a decoding step which returns a sequence of symbols given by the identifiers.

4.4 The Batch Generator

Responsible: Alexander Johansen (s145706)

At this point the data has been loaded into memory by the data loader and is simply waiting to be fetched and used for training. The batch generator is responsible for delivering well formatted batches of data samples from the entire dataset. An instance of the `BatchGenerator` is used as a generator, that yields batches continuously w.r.t. an iteration schedule (see section 4.4.1). Each batch is built from a set of samples in accordance with the iteration schedule.

The two main methods that are defined in the `BatchGenerator` class are `gen_batch()` and `_make_batch()`. The former picks individual samples from the dataset and uses the latter to format the data into numpy arrays which it then yields to the caller. The formatted batches are delivered as key-value pairs. A batch contains NumPy arrays [van der Walt et al., 2011] of data for every accessible key. The keys and their values are as follows

`x_encoded` Encode the respective input sample w.r.t. the alphabet (see section 4.3)
`t_encoded` The same as above but for target sample
`t_encoded_go` As `t_encoded` with a start-of-sequence symbol prepended
`x_spaces` Indices pointing to the spaces for the respective input sample
`x_spaces_len` Defines the length of the previous lists
`x_mask` List containing 1's with the desired size for each input sample and the rest is 0
`t_mask` The same as above but for target sample
`x_len` Defines length of respective sample (will no exceed predetermined maximum size)
`t_len` The same as above but for target sample

We refer to the source/input samples as `x` and the target samples as `t`, which is in accordance with the literature and the sections presented in chapter 2. Moreover, the `_make_batch()` method adds a symbol that indicate the beginning of sequences such that the model is for instance able to distinguish if it has reached a new sample. In section 5.3.1.1 we will describe how a batch is used in our models.

4.4.1 Iteration Schedule

In order to intelligently combine the individual samples into batches for training, we use iteration schedules. Iterations schedules are implemented as generator functions where for every invocation it yields a list of indices, that points to a set of training samples to include in the next batch. Python enables us to flexibly substitute one iteration schedule for another or even wrap one iteration schedule inside another.

When training a model using stochastic gradient descent, there are a lot of considerations to take into account when grouping the samples into the minibatches that are used.

First of all, since we're using a dynamically unfolded RNN, we can save a lot of time and memory by ensuring that the lengths of samples within a single batch do not vary too

much. If some samples in a batch are shorter than others, they have to be zero-padded so they are all equally long. Even if they do not actually affect the predictions, the zero paddings will take up space in the memory and computations. Thus it is possible to save time and memory by making batches where all samples have almost the same length.

Another thing to keep in mind when making batches is that they should be unique between epochs. That is each sample in the training set should preferably be batched with different samples each epoch.

Here we will walk through a few different iteration schedules that we have implemented and explain how they work and what function each of them serves.

4.4.1.1 Bucket Schedule

The bucket schedule is built with the goal of creating batches where the inputs x for all samples have approximately the same length and where the targets t for all samples have approximately the same lengths. This solves the problem, discussed above, with wasting computational resources on zero padded values.

The bucket schedule function works by constructing a tuple for each sample in the training set and collecting the tuples in a list. Each tuple contains the sample's original index, so it can be easily retrieved and added to a batch, and an auxiliary value for roughly sorting the sample-tuples w.r.t. both their input x and target t sequence lengths.

It is possible to consider this importance value as determining into which buckets each sample is placed into. The auxiliary value that is used for sorting the list is computed for the i$^{\text{th}}$ training sample using the following formula:

$$\text{aux}_i \leftarrow k \cdot \left\lfloor \frac{T_x^i}{\text{fuzziness}} \right\rfloor + \left\lfloor \frac{T_t^i}{\text{fuzziness}} \right\rfloor, \tag{4.3}$$

where fuzziness is a small positive integer which will be discussed later, and k must be some large value to make sure that the samples are always sorted on the source sequence length and the target sequence lengths are merely used for breaking ties.

In our implementation we defined $k = 16{,}384$, because we could perform the multiplication efficiently with a bitwise left shift of 14. Moreover, the T_x^i and T_t^i are the respective lengths of the source and target sequences of the i$^{\text{th}}$ sample.

Sorting the list of tuples according to these auxiliary values yields a list of indices where samples that are in near proximity to each other are approximately of the same sequence length for both input x and target t. Because of this, any contiguous slice of samples from the list can be used as a batch where the samples have almost the same lengths. Thus we define the individual batches by slicing the list into chunks according to the `batch_size` parameter.

Finally we usually serve these batches in randomised order such that the samples in contiguous batches are likely to vary in lengths. If we do not shuffle the batches, the model will be trained on the smallest samples first and the longest batches last during

each epoch. This would generally lead to inferior performance because the model would likely perform well when making predictions on samples similar to those it has been trained on recently, and poorly on all other samples. We did however try starting each training run without shuffling the batches for the first ~10k iterations. (See section 4.4.1.3 below.)

For every epoch this list will define the indices of training samples to fetch. It is possible to shuffle the list such that it varies from epochs. We define this list of tuples as the iteration schedule because it defines the order in which training samples are fetched for iterations. Moreover, for every epoch we permute the range of possible batches to fetch samples in different order for different epochs. Thus we are not guaranteed to get batches of ascending order but they may come in a arbitrary order.

Having equal auxiliary value is an equivalence relation, that we use to partition the samples into equivalence classes. In some sense the auxiliary value is used to partition the samples into small buckets of samples that have near equal sequence lengths.

The fuzziness parameter and shuffling with stability. The definition of the auxiliary value in eq. (4.3) above includes a fuzziness parameter that needs explaining. In some sense the fuzziness parameter dictates the size of each of the buckets that the scheduler has taken its name from.

Dividing by the fuzziness and rounding down is equivalent to performing integer division by the fuzziness. This means that if the fussiness is e.g. 3, then sequences of length 6, 7, or 8 would give the same result. So by increasing the fussiness we make it more likely that different samples are assigned the same auxiliary value (and thus end up being in the same "bucket") even if their lengths differ slightly.

When we make the batches, we generally shuffle the samples before sorting them with a stable sorting algorithm. This ensures that samples that have the same auxiliary values are ordered randomly (amongst each other) in the sorted list. Thus slicing the list into batches is unlikely to yield identical batches between epochs.

On one hand increasing the fuzziness parameter results in samples with variety of their sequence lengths receiving the same auxiliary values, and thus being grouped together and put in batches together, increasing the need for zero padding. On the other hand, when more samples have the same auxiliary values, they are more likely to preserve more of the ordering from the random shuffling after the stable sorting, which results in more diverse batches between epochs.

Thus choosing a good value for the fuzziness parameter is a tradeoff between increasing variety in the batches between epochs, and reducing the need for the zero paddings that lead to wasted computations. We generally set the fuzziness parameter to have a value of 3. It is worth keeping in mind that the fuzziness parameter is used on both the input and target sequence lengths, so the number of "buckets" decrease like the inverse of the square of the fuzziness parameter.

4.4.1.2 Variable Bucket Schedule

During our experiments we found that the model can be constrained by the available RAM (Random Access Memory) on the GPUs, especially when training the models with long sequences. However when using the bucket schedule, described in section 4.4.1.1 above, it is only the batches with the longest sequences that get anywhere near utilising the full memory capacity of the GPUs. Because we use a dynamic unfolded RNN, described in section 4.8.2, the batches will only take up memory roughly proportionally to the sequence length of the largest sample in the batch during training which is wasteful.

In order to better take advantage of all the available memory on the GPUs during training, we built the *variable bucket schedule*. In addition to avoiding mixing samples of different lengths in the same batches (like the regular bucket schedule), the variable bucket schedule will also increase the number of samples in batches that contain short sequences and decrease the number of samples in batches with long sequences.

The variable bucket schedule takes a *threshold* parameter, τ, as input instead of a fixed target batch size. The schedule will then cram as many samples into each batch, B, as it possibly can without violating the following condition:

$$\left(\max_{s \in B} T_x^s + \max_{s \in B} T_t^s\right) \cdot |B| \leq \tau, \tag{4.4}$$

where T_x^s and T_t^s are respectively the input and target lengths of the sample s. The threshold, τ, is set to be as large as possible while still avoiding any "out of memory" errors, so we typically end up with values on the order of 100,000.

The variable bucket schedule achieves this by making a fuzzy-sorted list of indices using the auxiliary values defined in eq. (4.3) just like the regular bucket schedule does, such that samples that have similar lengths are near each other in the list. The schedule then starts adding samples from the top of the list to the current batch until the condition in eq. (4.4) would be violated by adding the next sample. When that happens the current batch is finalised and a new empty batch is created to which the next sample is added to instead. When all samples from the list have been put in batches, the batches are yielded in random order.

Besides just enabling the models to train on more data in the same number of iterations by fully utilising the available RAM on the GPU, the variable bucket schedule also has another advantage. Having more samples or time steps in each batch leads to better predictions because each update of the parameters is less effected by the peculiarities of each sample. The graph in fig. 4.3 shows how the variable bucket schedule compares to the regular bucket schedule on the validation set during training.

4.4.1.3 Warm-up Schedule

Based on experiments we performed we found that the models would learn quicker if we made sure that the initial batches were composed of short sentences. The intuition is that it is easier to learn some simple representation first and then later scale up to

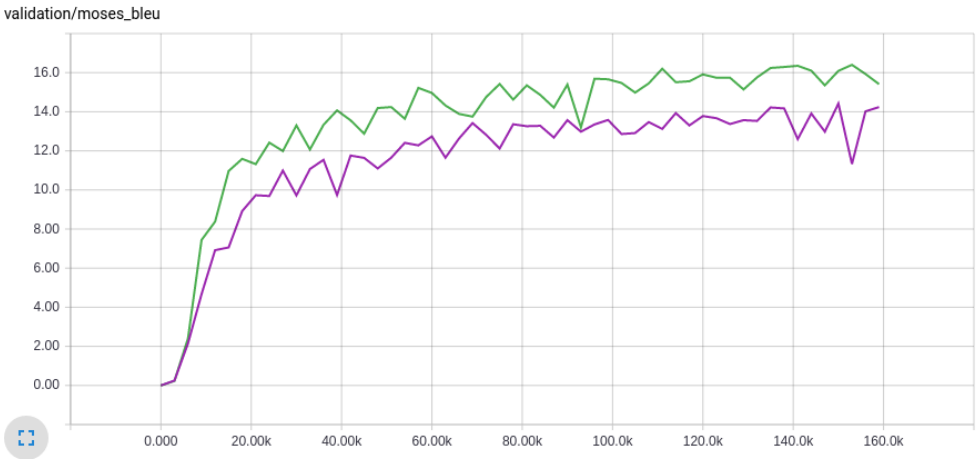

Figure 4.3: Graph comparing the BLEU scores of the same model with different iteration schedules on the validation set. The purple line represents the model that uses the regular bucket schedule while the green line represents the model that uses the variable bucket schedule.

learn more complex representation with some a priori knowledge gained from the simple representation.

We implemented a schedule to incorporate exactly this functionality. This warm up schedule uses the previously described schedule functionality to yield a list of indices sorted in ascending order w.r.t. the training sample's sequence length. Thus, no shuffling or fuzziness is added to the constructed list of indices.

This warm up schedule is invoked some defined number of iterations, where after the regular iteration schedule is invoked for the remaining training.

4.5 Configurations

Responsible: Jonas Hansen (s142957)

The model itself along with hyperparameters and other settings are defined in configuration files that are stored in the `configs` directory in the code repository. The configurations are implemented as classes that all provide a common set of methods and some basic properties. This enables the training script described in section 4.6 to interface seamlessly with different configuration files without having any specific knowledge or assumptions about them. The configuration file needs to provide the following for the training script to use:

1. Some variables that are used by the training script to specify basic settings; other basic settings include the frequency with which various log files should be update or status reports should be written to the console
2. Methods that define placeholders, model inference, prediction, loss and accuracy

3. Methods that yield batches of training or validation data

Basic Settings

The most important of the basic settings is the name that may be used to identify the configuration and is used to decide where to save the files that contain results, logs, and parameter checkpoints. Other basic settings are used to decide how often logs and parameter checkpoints should be saved, and how often to run validation and write status updates to the screen during training.

Model-Defining Methods

Placeholders are used to define the possible inputs for the model, and they are also defined in the configuration because we may want to give different inputs to new models when we try out new things. Of course we may also want to make changes to the inference, prediction, or loss+accuracy, so those are also defined by the configuration.

Data-Providing Methods

Finally, a configuration must provide methods that yield training or validation data that is preprocessed and formatted so it is ready to be used as input for the model. The training script can use these methods to get a batch of data and send it to the model without knowing any specifics about either the data or the model itself. The configuration generally does not have to contain a lot of code in order to provide these methods. Most of the hard work is taken care of by the excellent data loader and batch generator (which are described in detail in sections 4.2 and 4.4, respectively).

Large Default Model with Small Configs for Experimenting

This may seem like a lot of settings to define in a configuration file, but a typical configuration is very short and inherits almost everything it needs from the default base config. A typical configuration for testing some specific value of a hyperparameter may consist of only a few lines of code for overwriting a few variables from its super class. On the other hand this setup allows us a lot of flexibility to make new and bold experiments for two reasons:

1. We can easily define new configurations whenever we need to. If we wish to try a slight variation on some model, we can just make a new model that inherits from it and only have to rewrite the parts we want to be different.
2. When writing new configurations that inherit from the old, we don't have to worry about making mistakes when overwriting working models. We can quickly make configurations for trying out many different changes, and then use what we learned to update the default base configuration when we find out what works well.

This for instance makes it easy to define and train models with large architectural changes simply by defining a new configuration. To demonstrate how simple a configuration file may be we provide an example of a small config in listing 4.2, which is based on the default model in listing 4.1. It simply overwrites the batch size and everything else is inherited from the default model.

Listing 4.1: (`configs/default.py`) Settings of the default `Model` class.

```
class Model:
    # settings that affect train.py
    batch_size_train, batch_size_valid = 100000, 128
    seq_len_x, seq_len_t = 50, 50
    name = None       # (string) For saving logs and checkpoints. (None to disable.)
    visualize_freq = 10000  # Visualize training X, y, and t. (0 to disable.)
    log_freq   = 100        # How often to print updates during training.
    save_freq  = 1000       # How often to save checkpoints. (0 to disable.)
    valid_freq = 500        # How often to validate.
    iterations = 5*32000    # How many iterations to train for before stopping.
    train_feedback = False  # Enable feedback during training?
    tb_log_freq = 500       # How often to save logs for TensorBoard
    # datasets
    train_x_files = ['data/train/europarl-v7.de-en.en.tok',
     ↪  'data/train/commoncrawl.de-en.en.tok',
     ↪  'data/train/news-commentary-v10.de-en.en.tok']
    train_t_files = ['data/train/europarl-v7.de-en.de.tok',
     ↪  'data/train/commoncrawl.de-en.de.tok',
     ↪  'data/train/news-commentary-v10.de-en.de.tok']
    valid_x_files = ['data/valid/newstest2013.en.tok']
    valid_t_files = ['data/valid/newstest2013.de.tok']
    test_x_files  = ['data/valid/newstest2014.deen.en.tok']
    test_t_files  = ['data/valid/newstest2014.deen.de.tok']
    # settings that are local to the model
    alphabet_src_size = 310  # size of source alphabet
    alphabet_tar_size = 310  # size of target alphabet
    alphabet_src = Alphabet('data/alphabet/dict_wmt_tok.de-en.en', eos='*')
    alphabet_tar = Alphabet('data/alphabet/dict_wmt_tok.de-en.de', eos='*', sos='')
    char_encoder_units = 400  # number of units in character-level encoder
    word_encoder_units = 400  # no. units in word-level encoders (for- and backward)
    attn_units  = 300         # no. units used for attention in the decoder
    embedd_dims = 256         # size of character embeddings
    learning_rate, reg_scale, clip_norm = 0.001, 0.000001, 1
    swap_schedule = { 0: 0.0 }   # data argumentation schedule (0.0 to disable.)
    schedule_kwargs = { 'fuzzyness': 3 } # arguments (kwargs) for scheduling function
```

4.6 Training Script

Responsible: Elias Obeid (s142952)

When we want to train (or continue training) a model we use the training script, `train.py`. The training script takes the path to a configuration module (see section 4.5)

Listing 4.2: (`configs/batch-size/size-1024.py`) Overwriting a few configs from base `Model` class (listing 4.1) to experiment.

```
1 from configs import default
2 class Model(default.Model):
3     # overwrite config
4     name = 'batch-size-1024'
5     batch_size = 1024
```

as argument and loads the specified configuration module and uses it to define and train the model. The training script is responsible for almost all functionality that is not provided through the configuration file, i.e. anything that is always needed no matter which model we might want to train. The training process continues until the iteration limit has been reached, which is give by the `iterations` variable from the configuration.

When the training script runs, one of its first tasks is to make calls to the appropriate methods, that are provided by the configuration, for setting up the computation graph that defines the model itself and for starting to load and prepare data for training and validation.

After the computation graph for the model has been defined, the training script will initialize all weights and biases of the model. The parameter initialization is done either by restoring the parameters from a previously saved checkpoint file, by looking for one with the name defined by the configuration, or by generating new, pseudo random values based on the model definition. Additionally, the training script will make the necessary preparations for saving parameter checkpoints during training.

A very central functionality that is handled by the training script is the basic training loop that retrieves batches of training data using the methods that are provided by the configuration, and feeds them to the computation graph. Some settings for this training loop is provided by the configuration, including the total number of iterations before the training should be terminated, and how often status updates should be printed.

With a certain frequency (which is again specified by the configuration file), the training will be paused so the model can be validated on the validation set instead, to provide an impression of the models ability when given a dataset that is separate from the training set.

Also based on the name that is defined in the configuration file, the training script will append log files with tensor summaries during training that can be opened and viewed through TensorBoard to keep an eye on how the training is progressing.

4.7 Utilities

Responsible: Alexander Johansen (s145706)

This section holds our custom BLEU implementation (section 4.7.1) and our data augmentor (section 4.7.2).

4.7.1 BLEU Implementation

As mentioned in section 2.9.2, the bilingual evaluation understudy (BLEU) is a useful and popular technique for evaluating the quality and accuracy of translations. However, the formal definition of BLEU does leave some unfortunate edge cases, where the algorithm breaks down and gives non-intuitive scores for translations.

Recall from eq. (2.75) (which is repeated here in eq. (4.5) for convenience), that BLEU is computed as

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log(p_n)\right), \tag{4.5}$$

where p_n are the modified precision scores defined in eq. (2.74).

If the modified precision score is 0 for just one n, then BLEU is undefined because $\log(0)$ is undefined, and we observed some implementations that would just give a BLEU score that was close or equal to 1.

Alternatively, since

$$\log(x) \to -\infty \text{ for } x \to 0, \tag{4.6}$$

the hypotheses could be penalised infinitely and given a BLEU of 0, but that does not seem fair either, because even getting all the 1, 2, and 3 grams correct would result in a BLEU score of 0 if there are no matching 4-grams.

For this reason, the open source implementations of BLEU that we tested produced unintuitive and confusing answers during the first phases of the project, when our models were not as good and we were constricting our selves to only work on shorter sentences. Thus we decided to make our own small-scope implementation of BLEU, covering just the features we needed at the time.

Our own implementation differs from other implementations mainly by being slightly more forgiving in some cases when the sentences are very short. For example, a hypothesis sentence will only be penalized on the number of n-grams if the reference sentence contains at least n words.

In the later stages of the project, in addition to our own implementation, we started using MOSES' multi-bleu implementation, which is written in perl. Our experiments show that outside of the edge cases mentioned above, our own implementation and MOSES' implementation mimic each other, so when one increases or decreases, the other does as well. However our own implementation is generally optimistic, and it reports scores that are consistently about 0.02 higher compared to MOSES' implementation. This means that while our own implementation is very useful for measuring the accuracy and quality of translations, it cannot be used for comparing results with others who have used standard implementations such as MOSES'.

In order to interface with the MOSES multi-bleu perl implementation from our own training script, which is coded in python, we made a function that calls the perl script through shell commands. Before the MOSES multi-bleu script is called, the reference and predicted sentences that should be used to compute the BLEU score are saved to text files on the storage drive. The MOSES multi-bleu script is then called and the paths to the reference and prediction files are given as arguments. The output from MOSES multi-bleu is retrieved as a string which is parsed for the needed information (i.e. the BLEU score) to be passed on.

4.7.2 Data Augmentation

In the attempt of adding robustness to the decoder we implemented the `Augmentor` class. Its purpose is to transform and add noise to the decoder's input values.

The class' constructor takes one optional argument, which points to the function to be used to transform the given data. Data transformation is performed by invoking the `run()` method, which is attached to the `Augmentor` instance.

4.7.2.1 Transformer Functions

We implemented two transforming functions, which we will present in the following paragraphs. Note, we decided not to transform elements with a size of less than five.

Swap Neighbouring Elements by Chance. We implemented a method for swapping neighbouring elements in the given list. The method (`swap_chars_by_percent()`) trivially swaps the given element with the element immediately to its right. This introduces some form of spelling mistakes.

The number of swaps performed is computed as the augmentation amount (which was given as an argument to `run()`) multiplied by the last element we want to be swapable.

$$\text{idx}_\top \leftarrow \text{size}_i - 2 - \text{skip}_\leftarrow \tag{4.7}$$

$$\text{idx}_\bot \leftarrow \text{skip}_\rightarrow, \tag{4.8}$$

where $\text{idx}_\top$ and $\text{idx}_\bot$ define the upper and lower limits of swapable elements, size_i gives the size of the i^{th} element, and $\text{skip}_\leftarrow$ and $\text{skip}_\rightarrow$ defines how much to ignore from the end and beginning of the elements list (these are by default set to 0, and otherwise given to `run()` as arguments). By default we ignore the last two elements because the very last element is an end-of-sequence symbol (which we do not want to move) and the method simple swaps with the element to the right of the one being swapped, thus we ignore the last two elements to not swap with the end-of-sequence symbol. We then compute the number of wanted swaps as

$$\text{swaps} \leftarrow \begin{cases} \lceil \text{idx}_\top \rho \rceil & \text{if } \rho \text{ is defined} \\ 1 & \text{otherwise} \end{cases}, \tag{4.9}$$

where ρ defines the amount of wanted augmentation $[0, 0.4[$. The next index to swap i is computed as

$$i \leftarrow \xi(\text{idx}_\perp, \text{idx}_\top), \tag{4.10}$$

where ξ is a function defined as

$$\xi(\text{lower}, \text{upper}) = \lfloor \zeta \cdot (\text{upper} - \text{lower} + 1) \rfloor + \text{lower}, \tag{4.11}$$

and ζ yields a random value from the continuous uniform distribution of the interval $[0.0, 1.0[$.

Finally, the i^{th} element is thus swapped with the $(i+1)^{\text{th}}$ element. Furthermore, we maintain a set of indices that have already been swapped and their immediate neighbour. We do not wish to swap individual elements multiple times. If an element has already been swapped, we find another element to swap.

Transform Elements with Probability. We implemented another method for transforming a given element in the list of elements by a some variable probability. The method (`chars_to_noise_binomial()`) performs a random swap of an element with another element from the same list of elements by a binomial distribution. Note that this other element may be of the same value as the original element.

The upper and lower limits are computed as

$$\text{idx}_\top \leftarrow \text{size}_i - 1 - \text{skip}_\leftarrow \tag{4.12}$$

$$\text{idx}_\perp \leftarrow \text{skip}_\rightarrow, \tag{4.13}$$

which is almost identical to the above method, but now we only ignore the end-of-sequence symbol.

For every element in the list of elements we construct a list, χ, of $|\text{idx}_\top|$ elements containing either 1s and 0s from a binomial distribution. This list is constructed from a binomial distribution with one trial, and ρ as the probability of success.

Now, for every element in χ if the i^{th} element is 1 the i^{th} element of the given list of elements is swapped with a random element from the same list, computed by $\xi(\text{idx}_\perp, \text{idx}_\top)$.

4.7.2.2 Transforming Data

The `run()` method is invoked with a list of elements to transform, a list of each element's size, and the amount of noise to add. Moreover, two optional parameters can be passed to the method which define how much to ignore from each end of the given data. We assume that the list of elements contain lists.

We decided to put a limit on the augmentation amount to avoid unreadable pieces of data. We agreed that the mount should be < 0.4 to still produce usable results.

4.8 Customised Functionality

Responsible: Jonas Hansen (s142957)

Previously, we mentioned that we used the TensorFlow machine learning framework to build and train our models. We also described that the framework had a few tools that we were handy, e.g. TensorBoard to view training progress. However, as it is a fairly young framework some of the higher order implementations had very limited functionality, e.g. the recurrent neural network and encoder-decoder libraries. We found it to be tightly coupled and left very little room for customised solutions, which we very much needed. Furthermore, some functionality is not at all supported in TensorFlow yet. For this reason we have implemented a few customised algorithms for use in this project.

This section presents and describes our custom implementations for loss function for sequence to sequence (encoder-decoder) library (section 4.8.1), a dynamic recurrent neural network library (section 4.8.2), the encoder-decoder architecture with the attention mechanism, and functionality for gathering of scattered elements (section 4.8.3). Other interesting contributions to the framework are presented in section 4.8.4.

4.8.1 Sequence-to-Sequence Loss Function

To compute the loss over our predictions we use multinomial cross entropy (elaborated in section 2.3.2.1) over all predicted output classes for each point in time along a sequence of characters. This is also known as sequence-to-sequence (seq2seq) loss and is implemented by TensorFlow in their seq2seq library[1]. However, as we found that TensorFlow's implementation had some constraints and computational issues, we felt it necessary to develop our own implementation.

The framework's built-in `sparse_softmax_cross_entropy_with_logits()` function computes the seq2seq loss of some predictions w.r.t. the expected targets. It is built to take two-dimensional (2D) tensors and compute the loss of these. The TensorFlow implementation stores the given predictions, along with the targets, in lists. These lists are thus made up of 2D tensors, which are then fed into the function to compute the loss for each prediction individually.

We identified one key issue with this implementation; the loss function must be invoked as many times as there are 2D tensors in the lists. Being aware of a possible performance issue with this approach we sought out an approach that would invoke the loss function only once.

We used TensorFlow's `sequence_loss()` as inspiration to build our own version, which we defined as `sequence_loss_tensor()` in `utils/tfextensions.py`. Utilising TensorFlow's built-in functionality to handle different shapes of tensors our implementation takes a 3D tensor and reshapes it into a taller 2D tensor. We are thus able to invoke the built-in loss function once and work with the result to compute the average loss for the entire set of tensors.

[1] `https://github.com/tensorflow/tensorflow/blob/63b7a9807390e980e620d11d0bb2d179685d24fa/tensorflow/python/ops/seq2seq.py` — the URI points to the framework's seq2seq library at the time of writing.

To give an example of our thought process consider a time step, h_t, in an RNN sequence. It will often have the form $h_t \in \mathbb{R}^{n \times m}$, where n represents the batch size (the height) and m represents the number of hidden units (the width) in the RNN cores. To represent a collection of time steps we identified two approaches:

1. Store sequences in lists of 2D tensors (TensorFlow's approach) to represent each time step, i.e. $[h_1, h_2, \ldots, h_t]$. Now iterate over the list and evaluate each tensor's loss individually.
2. Store sequences in 3D tensors (our approach), i.e. $h \in \mathbb{R}^{n \times t \times m}$, where t is the sequence length. Reshape the 3D tensor into a 2D tensor, which yields a tensor $h \in \mathbb{R}^{nt \times m}$. Evaluate the tall 2D tensor in one go.

We did not explicitly investigate the possible performance gain from the above mentioned manoeuvre, but we empirically noticed a significant performance boost after the implementation. We believe that this boost is due to TensorFlow utilising various optimisations for mathematical functions, e.g. from the BLAS library, and a single invocation with a taller matrix would avoid a bottleneck of multiple calls to the loss function.

4.8.2 Dynamic Recurrent Neural Network

To compute our conditional predictions we use an encoder-decoder architecture as described in section 2.7.4. Such an architecture requires an implementation of a recurrent neural network (RNN). More specifically, for the encoder we need an implementation of an RNN with a gated recurrent unit (GRU) core. For the decoder we need a modified RNN that can handle feeding predictions to the next time step as explained in section 2.7.4.

TensorFlow already exposes such implementations in the RNN library[2]. However, these have some deficiencies; their implementation is verbose and it unfolds the network and requires all sequences to have a certain length. We believe that the implementation is verbose because they may have to consider legacy code that has been implemented previously. The requirement of all sequences to have a certain length is not a computational issue for the standard RNN implementation, as it only computes the necessary time steps. On the other hand, the decoder architecture computes all time steps (including irrelevant padding). This is obviously a costly operation as all sequences are required to decode the maximal sequence length. At the time of writing there were no dynamic implementation of an RNN that suited our needs, which could overcome this extraneous computation.

The task at hand is a computationally expensive one. Therefore, we agreed to develop a flexible and customised library containing RNN, GRU, and attention functionality. Throughout the project we used TensorFlow's implementation as inspiration.

[2] https://www.tensorflow.org/versions/r0.9/api_docs/python/nn.html#recurrent-neural-networks

We used TensorFlow's `TensorArray`[3] and `while_loop()`[4] operations to build the functionality needed for our custom RNN library. We aimed for a minimal and compact implementation suited specifically for our needs. While performing various experiments we empirically verified that our custom library gave a significant performance boost compared to TensorFlow's libraries.

Another large advantage of using a dynamic recurrent network architecture is that it only uses the amount of RAM that is actually needed for batches of short sequences. This enabled us to speed up the training further and fully utilise the available ram by implementing the variable bucket schedule. (Section 4.4.1.2)

Even though the RNNs that are used for encoding and decoding the sequences are much alike from a general perspective, we built two different functions for adding RNNs to a model. The `encoder()` is best suited for adding RNNs on the encoder side, while `attention_decoder()` is obviously made for adding the decoder to a model. These RNN functions both use GRUs and obviously have a lot of code in common.

Encoder. The `encoder()` function takes four arguments (all mandatory) and returns the encoder's final hidden state as well as a tensor with all hidden states. The four mandatory arguments are `inputs`, `lengths`, `name`, and `num_units`. The `lengths` defines the input sequences' lengths from which the encoder can deduce the maximum and minimum sizes. TensorFlow has named variable scopes which organizing the various operations that make up the constructed computational graph, and the `name` argument is used to define such a named scope.

The `num_units` arguments defines how many units to use for the GRU core i.e. the amount of computational units.

The vectors representing the previous state and the new inputs for the RNN itself are concatenated. This enables us to handle the multiplications with their respective weights in one fell swoop by combining their weight matrices as well. Thus, the encoder's weight matrices are $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h \in \mathbb{R}^{(i+n)\times n}$, where i is the number of input features (the last dimension of the `inputs` tensor) and n is the `num_units`. The bias vectors are $\mathbf{b}_z, \mathbf{b}_r, \mathbf{b}_h \in \mathbb{R}^n$.

The hidden state of the RNN is initialized with a zero-vector.

Attention decoder. For the decoder in our models to give the best results they need an attention mechanism to help decide which part of the outputs from the encoder that best serves as context for the next output that the decoder should generate.

The `attention_deceoder()` takes ten arguments (nine mandatory), and returns a tuple of four objects. The `attention_decoder()` function technically builds two different RNNs that share weights. The first RNN is for training the network, while the second RNN is for validating/testing. The difference is that the first RNN uses the

[3] https://www.tensorflow.org/versions/r0.9/api_docs/python/tensor_array_ops.html#TensorArray

[4] https://www.tensorflow.org/versions/r0.9/api_docs/python/control_flow_ops.html#while_loop

`target_input` and `target_input_lengths` arguments to feed the target for prediction with a lag of 1 time step, which helps it learn to make correct predictions faster. The second RNN uses it's own output from the previous time step along with the weights and biases `W_out` and `b_out` to predict embedding (using `embeddings`) of the most likely character from the previous step and use that as input instead of the actual true labels from the training data.

The first two objects are the RNN's final state, a tuple of all states for the training RNN and the last two objects are the validation RNN's outputs and validation attention tracker that we use for debugging and visualising the attention. The decoder's optional argument is a name for a TensorFlow variable scope like for the encoder.

The implementation of the attention mechanism for the decoder is coded by closely following the instructions which are laid out by Bahdanau et al. [2014]. It creates a context vector, which is a linear combination of all the elements of the `attention_input` argument, where the weights for the combination are chosen based on the decoder RNN's current state. We refer back to the theoretical discussion in section 2.7.4 for details on this. The `attention_lengths` argument is used for masking so the decoder does not put attention on values that represent zero-padding instead of actual text. The `num_attn_units` argument dictates how many units to use for generating the weights for the linear combination.

The GRU is very similar to those from the `encoder()`, but they take the context that is generated by the attention mechanism as input as well. Thus the context vectors are concatenated with the vectors representing the previous state and the new inputs. The weight matrices are $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h \in \mathbb{R}^{(i+2\cdot n)\times n}$ where, like before, i is the number of input features (the last dimension of the `inputs` tensor) and n is the `num_units`. The bias vectors are $\mathbf{b}_z, \mathbf{b}_r, \mathbf{b}_h \in \mathbb{R}^n$.

In contrast with the `encoder()` RNN, the hidden state of the RNN is initialized by the final state of the encoder, which is given as the `initial_state` argument.

4.8.3 Grid Gather

As mentioned in section 1.4.1, character-based machine translation is more challenging than making a purely word-based model. This is because each word consists of multiple characters, and thus the model will have to keep the memory of what it has already parsed or written over much longer sequences.

In order to mitigate this problem many of our models use a mechanism where they, in addition to the *character-level* RNN, also process the inputs using a second RNN, which only takes a carefully selected subset of the outputs from the character-level RNN as its input. The subset of outputs that are processed by the second RNN is chosen so the elements represent the last character from each word in the input sentence. For this reason we usually refer to the second RNN as the *word-level* RNN.

When we use stochastic gradient descent, where a whole minibatch of data is processed at each iteration, the output from each the character-level RNN will be a tensor of rank

3 with the following size:

$$\text{batch size} \times \text{character sequence length} \times \text{features}.$$

For each sample in the batch we wish to pick out the feature vectors that correspond to the last character of each word, and put them in a new (smaller) tensor with size:

$$\text{batch size} \times \text{word sequence length} \times \text{features}.$$

This could easily be done with python, but it would not be useful to us, mainly because it would prevent TensorFlow from computing gradients w.r.t. any parameters that are used below the word-level RNN. So in order to handle the transition from outputs, given by the character-level RNN, to inputs for the word-level RNN we made a custom composition of TensorFlow operations that can pick out specific parts of a tensor.

The composition takes advantage of TensorFlow's built-in `gather()` operation, which already does something similar to what we want, but only for a single dimension. We decided to name the function `grid_gather()` and implement it to be used in a similar way, so the composition takes two arguments: `params`, and `indices`. The `params` argument is the tensor from which values should be picked from, which in our typical case is the tensor of outputs from the character-level RNN. The `indices` parameter is a tensor of indices that specify which values from the `params` tensor to pick for the output. Note that the indices are 0-indexed.

Figure 4.4 shows an example of how the `grid_gather()` function is used. In the example the indices point to the last character in each word in the `params` tensor so the indices in the first row are 2 and 6 because those are the (0-indexed) positions of "u" and "e" in the sentence "You are".

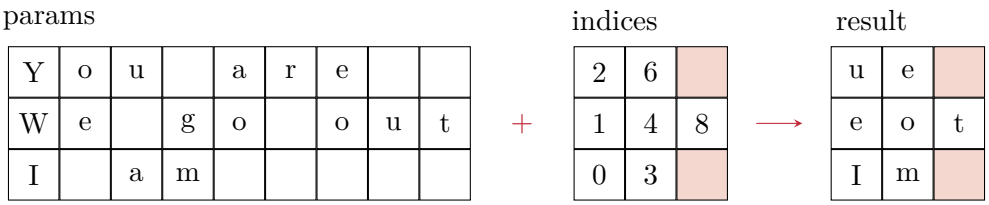

Figure 4.4: Example that illustrates what the `grid_gather()` function does. The values in the red-shaded cells are inconsequential because they will be ignored later on with masking, so they will typically just contain zero-padding. The values of the `params` and `result` tensors are shown as letters for illustrative purposes, in practice each of their cells would contain a feature vector.

4.8.4 Other Interesting Contributions

Here we present contributions to the TensorFlow framework that are not directly connected to our final product for this project. Nonetheless, we believe they are relevant

and show that we have been actively contributing to the framework. Moreover, on quite a few occasions we also edited some mistakes in TensorFlow's documentation as we were beginning the project. We could for instance fix spelling errors or provide a clearer description for some functionality.

4.8.4.1 User Friendly run Functionality

The TensorFlow framework has a `Session` class in which tensors and operations are executed and evaluated. To perform on step of computation of the given computation graph (tensors and respective operations) the class has a `run()` method[5]. By default TensorFlow's `run()` takes a `fetches` argument, which is a list of graph elements to evaluate for the step. It returns a list with the results of the computations for the respective elements. We added the possibility to give a dictionary instead of a list and the return value would also be a list of results (as the default implementation). We present this simple modification in listing 4.3.

Listing 4.3: Customised user friendly `run()` function.

```
1 def run(session, fetches, feed_dict):
2     """ Wrapper for making Session.run() more user friendly. """
3     if isinstance(fetches, dict):
4         keys, values = fetches.keys(), list(fetches.values())
5         res = session.run(values, feed_dict)
6         return {key: value for key, value in zip(keys, res)}
7     else:
8         return session.run(fetches, feed_dict)
```

The idea is now that instead of using a list to reference results we use the much more readable dictionary with strings as keys. Thus, assume that `res` contains the return value of `run()`. With TensorFlow's default implementation we may dereference the first value with `res[0]`, which may define the current loss value. If we use our own `run()` implementation and provide the `fetches` as a dictionary with relevant keys, we may for instance dereference the loss with `res["loss"]`, and the predictions are dereferenced with `res["ys"]` (by convention from many other publications). Using dictionaries is thus much more readable for the programmer, and the software will automatically be more maintainable.

[5] https://www.tensorflow.org/versions/r0.9/api_docs/python/client.html#Session

CHAPTER 5
Experiments and Results

In this chapter's section 5.1 we present the data we worked with. We will describe how we tested the data set, and present the different data sets we worked with. Moreover, we will discuss a few challenges we had. We present a more detailed description of the structure of our models in section 5.2 and how training the models were trained in section 5.3. Finally, in section 5.4 we present the different results we obtained from our experiments.

5.1 The Data

Responsible: Alexander Johansen (s145706)

Figure 5.1: Commander Data — Our favorite Data!

A very important aspect of training neural networks, and machine learning algorithms in general, is the dataset used to gather the experience for the learning algorithm to learn from. Both the size (number of individual samples) and the quality of the dataset can have a significant performance impact. A large dataset can help to prevent overfitting, and by preventing overfitting allowing more complex models to be built. More data may also help to better generalisation if many different samples are present because the model may be able to find more unique features for different classes. If the dataset contains too much noise the model might train slow and fail to converge as the data might not have enough structure, or it might learn the noise of the dataset instead.

In order to train a model like ours to learn to translate sentences we need aligned parallel corpora of text. We specifically need pairs of sentences where the source sentence is written in one language and the target sentence is written in another language, and the meaning of the two sentences (the sentiment) should be as close as possible.

For the datasets that we work with are typically provided with a pair of basic text files; one for the source language, and one for the target language. In these files, each line corresponds to a sentence that matches the equivalent line in the associated file.

Throughout our project we have used a sequence of different datasets, working from easy debugging sets to more advanced datasets used in current literature. The purpose of early datasets (the debugging sets) has been to empirically investigate our learning algorithm. In the following we will present the datasets that we have used, which are

- four different debugging datasets
- Europarl, English-to-Danish
- Europarl, English-to-French (used by some researchers)
- WMT'15[1], English-to-German (used by most researchers)

5.1.1 The Four Debugging Datasets

To test our implementations from chapter 4 we created four datasets to test the sequence-to-sequence models ability to learn from data.

The first debugging dataset called "normal" is a challenge meant to have the sequence-to-sequence model copy a narrow dictionary (numbers from 0 to 9 and spaces) from input to output. This tested our ability to compile a network, have it track gradients and optimise the network to copy states. An example of the data is illustrated below:

```
(input, target)
('838', '838')
('8 97', '8 97')
('4 2', '4 2')
('', '')
('1 15', '1 15' )
```

The second debugging dataset called "number words" is word translations of numbers in English to Danish. This is a simple test of the network's ability to translate from one language to another with a small fixed dictionary. An example of the data is illustrated below:

```
(input, target)
('eightthreeeight', 'ottetreotte')
('eight nineseven', 'otte nisyv')
('four two', 'fire to')
('', '')
('one onefive', 'en enfem' )
```

The third debugging dataset called "number words caps" is an extension of the second debugging dataset, but with input words being randomly upper-/lower cased, forcing

[1] http://statmt.org/wmt15/translation-task.html

the algorithm to learn different representation of source words. An example of the data is illustrated below:

```
(input, target)
('EiGHtThreEeIGhT', 'ottetreotte')
('eiGHT nINeSeveN', 'otte nisyv')
('fOuR TWo', 'fire to')
('', '')
('ONe oNEfiVe', 'en enfem' )
```

The fourth debugging dataset called "number words caps decoder" is an extension of the second debugging dataset, but with output words randomly being flipped to upper case. This forces the decoder to pick up on whether a upper-/lower case letter was given as input. An example of the data is illustrated below:

```
(input, target)
('eightthreeeight', 'otteTREotte')
('eight nineseven', 'otte NISYV')
('four two', 'FIRE to')
('', '')
('one onefive', 'EN enFEM' )
```

5.1.2 English-to-Danish Europarl

While building our model from the ground we have primarily used the Europarl dataset for testing for multiple reasons. First, the dataset is available in an English to Danish version which gives us the opportunity to manually see how the model makes translations between two languages. The group members are strong at both languages which allowed us to get a first hand impression of the quality and shortcomings of the model's translations. Second, the dataset is of fairly high quality compared to some of the other WMT datasets, so we did not have to worry much about challenges with data quality.

Intuitively, a model learns what it is trained on. When our model is trained on the Europarl dataset it learns the language that politicians use in the European Parliament, which is very formal. This language is not usually how everyday civilians talk to each other on the street. However, we did find the translations amusing on multiple occasions. Specifically, we used the Europarl V7 English-to-Danish translation (Europarl V7 will be elaborated below).

Further, we found that Europarl is a renowned dataset (part of the WMT'15, which will be mentioned below). Moreover, some Europarl datasets are being used in literature [Ling et al., 2015a].

5.1.3 English-to-French Europarl

The next logical step for our dataset was to move to Europarl V7 English-to-French translation, as the structure of the dataset is very similar to the Europarl V7 English-to-Danish and Europarl V7 English-to-French is being used in literature with benchmarks [Ling et al., 2015a], which allowed us to validate our model.

5.1.4 WMT'15

WMT is an abbreviation for the yearly *Workshop on Statistical Machine Translation*, which publishes a collection of curated text corpora datasets for attendants to use in order to be able to compare their results on a common dataset. The WMT is not a specific dataset on its own, but a collection of different officially published and renowned dataset providers, such as Europarl, UN corpus, News commentary, Wiki headlines, etc. WMT'15 supports language pairs across many different languages, all are based on English-to{German, Czech, Finnish, Russian, or French} and reverse (different years support different datasets and languages). We performed our WMT experiments with the WMT'15 English-to-German dataset, which seems most commonly used in the literature [Luong et al., 2015a, Chung et al., 2016]. The WMT'15 English-to-German dataset consists of approximately 4.7m sentence pairs and is made up of the following corpora:

Europarl V7[2] **[Koehn, 2005]** is a collection of translated proceedings from the European Parliament (about 2.1m sentences). In order to ensure that its members can participate on equal terms in their respective native languages the European Parliament employs professionals to translate all of their proceedings in real time. These translations have then been cleaned and aligned. The Europarl dataset contains in the vicinity of 2m samples for each pair of languages. The quality of the Europarl dataset is very high, probably because the translations were made on the fly by professional human translators, so the content of the sentences correspond very closely to each other.

Common Crawl[3] is a dataset collected by crawling web pages (about 2.4m sentences). A subset of the web pages are translated to multiple languages, and the parallel corpus that we are using has been made by using automated algorithms to decide which sentences in one language version of a web page corresponds to the sentences in another language version. For this reason the language pairs are often not aligned properly and the target sentence does not match the source sentence, and thus it is of poor quality. We made these observations by manually looking through the data set.

News Commentary V10[4] is a dataset that is compiled by aligning the commentary on news from Project Syndicate[5] (about 220k sentences), which is a project that

[2] `http://statmt.org/europarl/`

[3] `http://commoncrawl.org/`

[4] `http://www.casmacat.eu/corpus/news-commentary.html`

[5] `https://www.project-syndicate.org/`

provides access to news commentary, debates, and discussion equally for people around the world. The datasets' quality is high.

5.1.5 Preprocessing of WMT'15

To compare results with others we need to preprocess our datasets (train/valid/test) according to accepted standards. As most models on WMT'15 are word based, certain preprocessing steps for words are applied. We will follow the preprocessing steps of Chung et al. [2016]. These steps include normalising punctuation[6] and aligning different tokens properly, e.g. spaces before commas/parenthesis (known as tokenisation)[7]. Further, all sentences with source length of more than 250 characters and target length of more than 500 characters are removed.

We will report our results without normalising punctuation and tokenisation and with 250/500 sentence length removal for the Europarl data sets and with all preprocessing applied (normalising punctuation, tokenization, 250/500 sentence removal) on the WMT'15 data set.

This will make us inept to perform exact similarity measures with previous results on the Europarl data set, which we consider reasonable as it is only meant as a debugging data set. Our preprocessing on the WMT'15 data set will allow us to do exact benchmarking with Chung et al. [2016].

However, it is to be mentioned that one of our model architectures (section 5.2) is based on spaces, applying preprocessing which adds extra spaces might have an interference with our models performance.

5.1.6 Testing Overlapping Datasets

To validate that all of our datasets were disjunct, we used the following procedure to test the amount of duplicates across and within datasets. First, to measure the amount of duplicates within datasets. To find the total amount of data we execute `cat <file> | wc -l` and to find the number of unique data we execute `cat <file> | sort | uniq -u | wc -l`.

The results of these uniqueness tests are presented in table 5.2. The table indicates that only a small fraction of the datasets contains duplicates. Furthermore, we investigated duplicates across train/valid/test splits by dumping the unique command from before into separate files with `cat <file> | sort | uniq -u > <file_unique>`. The files can then be compared and the number of unique sentences may be determined with `cat <file_unique_train> <file_unique_test> | sort | uniq -d`. This resulted in a handful of duplicates, mostly short to one-word sentences containing countries. We thus conclude that our training sets have been downloaded properly.

[6] https://github.com/moses-smt/mosesdecoder/blob/ef028446f3640e007215b4576a4dc52a9c9de6db/scripts/tokenizer/normalize-punctuation.perl

[7] https://github.com/moses-smt/mosesdecoder/blob/5d5bf1885d8ec1a9f8482dc3839ae8f5bc293b60/scripts/tokenizer/tokenizer.perl

Table 5.2: Numbers illustrating the amount of duplications in the various datasets.

purpose	dataset	duplicates	unique / total
	Europarl: English to Danish		
train	europarl-v7.da-en.en	3.9%	(1,892,087 of 1,968,800)
	europarl-v7.da-en.da	3.6%	(1,897,971 of 1,968,800)
valid	devtest2006.en	1.2%	(1975 of 2000)
	test2006.en	0.1%	(1997 of 2000)
	devtest2006.da	1.1%	(1977 of 2000)
	test2006.da	0.2%	(1996 of 2000)
test	test2007.en	1.3%	(1974 of 2000)
	test2008.en	0.8%	(1984 of 2000)
	test2007.da	1.4%	(1972 of 2000)
	test2008.da	0.7%	(1986 of 2000)
	Europarl: English to French		
train	europarl-v7.fr-en.en	3.2%	(1,944,248 of 2,007,723)
	europarl-v7.fr-en.fr	3.1%	(1,945,618 of 2,007,723)
valid	devtest2006.en	1.2%	(1975 of 2000)
	test2006.en	0.1%	(1997 of 2000)
	devtest2006.fr	1.0%	(1979 of 2000)
	test2006.fr	0.1%	(1998 of 2000)
test	test2007.en	1.3%	(1974 of 2000)
	test2008.en	0.8%	(1984 of 2000)
	test2007.fr	1.2%	(1975 of 2000)
	test2008.fr	0.7%	(1986 of 2000)
	WMT'15: English to German		
train	europarl-v7.de-en.en	3.6%	(1,851,356 of 1,920,209)
	europarl-v7.de-en.de	3.5%	(1,853,606 of 1,920,209)
	commoncrawl.de-en.en	4.4%	(2,293,535 of 2,399,123)
	commoncrawl.de-en.de	0.1%	(2,399,105 of 2,399,123)
	news-commentary-v10.de-en.en	1.0%	(213,996 of 216,190)
	news-commentary-v10.de-en.de	0.6%	(214,959 of 216,190)
valid	newstest2013.deen.en		
	newstest2013.deen.de		
test	newstest2014.deen.en	0.1%	(2999 of 3003)
	newstest2014.deen.de	0.0%	(3003 of 3003)
	newstest2015.deen.en	0.1%	(2167 of 2169)
	newstest2015.deen.de	0.2%	(2165 of 2169)

5.1.7 Representing Data in the Model

From the model's point of view, each sample is represented as a sequence of characters from a fixed alphabet (see section 4.3). When initialising the parameters of the model, an embedding is also created for each of the characters in the alphabet (see section 2.8 for our theoretical treatment of embeddings). Before feeding the data to the model, we iterate over each character in each sentence and translate it to the integer id that corresponds to the embedding. If a sentence (on either the source or the target side) contains a character that was too rare to be included in the alphabet, it will instead be replaced by the id of a special `UNK` (unknown) character. This is rarely necessary when the alphabet has a few hundred characters, which is one of the advantages of working with characters instead of words because the vocabulary/alphabet can be kept rather small.

We work with one minibatch (with multiple samples) at a time both for speed and for regularisation. The data is fed to the model as matrices where each row corresponds to a single sample and each column corresponds to a time step/character. Thus each element in the input matrix is the integer id that represents the character that belongs in at the corresponding time step for that sample. (See the transition from (b) to (c) in fig. 4.2) In order to make sure that all rows of the matrix have the same number of elements, all sequences are zero padded until they are as long as the longest sequence.

After that, the integer ids of the input are replaced with their embeddings. (See the transition from (c) to (d) in fig. 4.2) This way the model ends up working with two sequences $\mathbf{x} = (x_1, x_2, \ldots, x_{T_x})$ and $\mathbf{t} = (t_1, t_2, \ldots, t_{T_t})$ of embeddings. Note, that the length of the source sequence, T_x of course can be different from the length of the target sequence T_t. The embeddings are updated along with all other parameters in the model during training.

Figure 5.3 visualizes the embeddings of a subset of the alphabet that we use when projected on a 2D plane using t-SNE [Van der Maaten and Hinton, 2008]. Note that related characters, such as lower-case and capital versions of the same letter, have a tendency to be close to each other. This shows that the embeddings of these characters have been updated during the training to be close together in the embedding space which means that they carry almost identical meanings to the model.

5.1.8 Challenges with Varying Sequences of Data

The data samples come in many lengths. If at least one of the samples in a minibatch is τ long then the full set of computations that are required to perform one time step must be performed for *all* samples in the minibatch up to time step τ. This may end up wasting quite a bit of processing time on time steps for samples where they are not used for anything. Especially if a few samples in each batch tend to be much longer than the rest.

A common solution to this challenge is bucketing where the samples are grouped according to their lengths. Then each minibatch is composed entirely of samples from a single

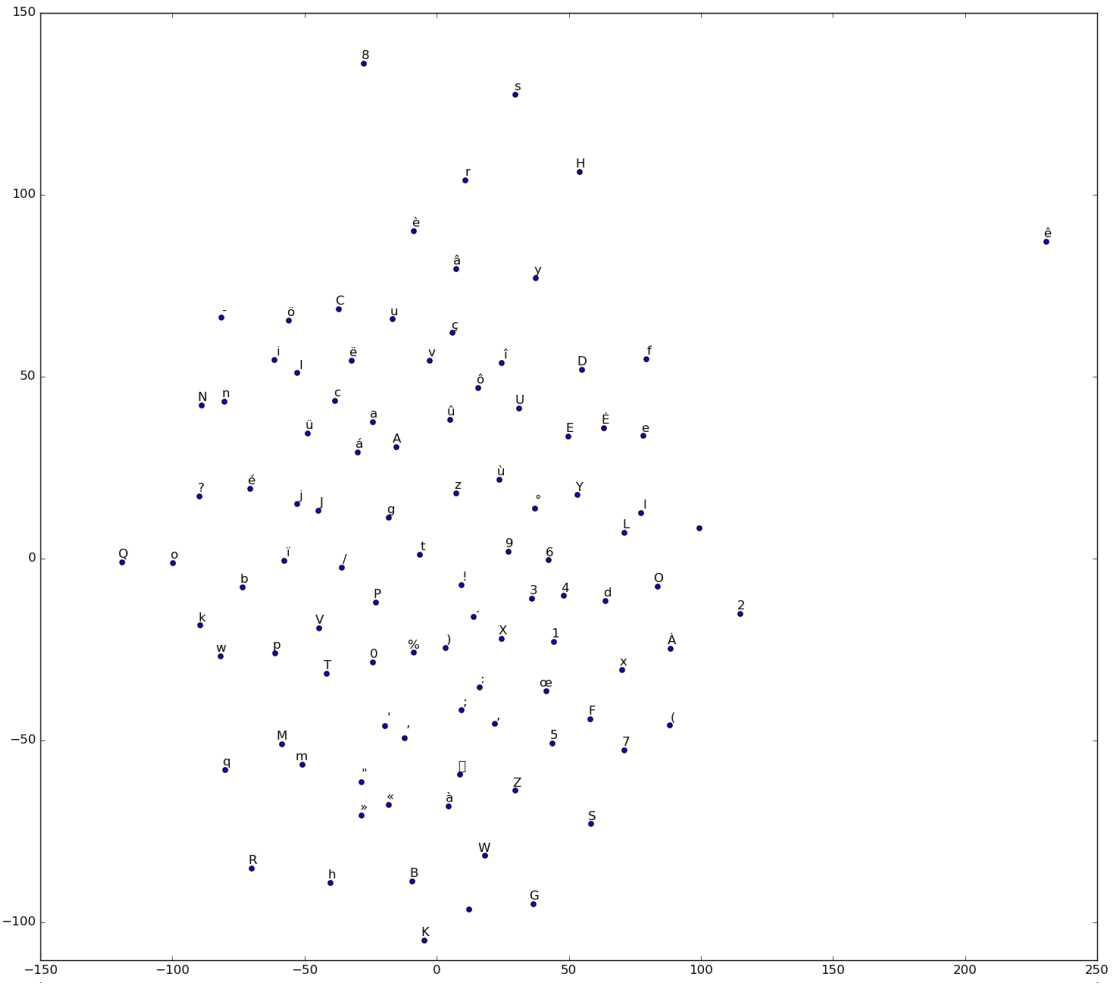

Figure 5.3: Illustration of character embeddings projected down on a 2D plane with t-SNE.

bucket. This of course reduces the amount of wasted computations but it also limits the possible combinations of minibatches that can be made. However, if each bucket contains enough samples then this is not a problem. Moreover, another challenge when using bucketing is how to make sure that the whole dataset has been trained on only once before starting from the beginning again. In order to overcome these challenges, we have implemented a new way of scheduling the samples for training. We have described the solution in details in section 4.4.1.

5.2 Model Architecture

Responsible: Jonas Hansen (s142957)

In section 4.8.2 we described our implementation of a sequence-to-sequence framework.

In this section we will provide the specific architecture that we used, given these abstract frameworks, to build our neural machine translation model. We will supply two different encoder models; the character-level encoder (section 5.2.1) and `char2word` encoder (section 5.2.2). In section 5.3 we illustrate these two encoders used in our models. Next, we will supply a decoder in section 5.2.3, and finally in section 5.2.4 we present the alignment model.

5.2.1 The `char` Encoder

Our character-level encoder (referred to as the `char` encoder) is built upon a bi-directional RNN using GRU cores [Cho et al., 2014c] for computing each time step (see section 2.7.4). In the following equations we remove the bias vectors from our formulations for readability. Our `char` encoder takes a source sequence $\mathbf{X}$ defined as

$$\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_{T_x}), \mathbf{x}_i \in \mathbb{R}^{\Sigma_{src}}, \tag{5.1}$$

where each symbol/character is represented by a `1-of-`Σ symbol vector. Σ_{src} represents the vocabulary size of the encoder's dictionary. The encoder's forward state is computed as

$$\overrightarrow{\mathbf{h}}_i = \begin{cases} (1 - \overrightarrow{\mathbf{z}}_i) \odot \overrightarrow{\mathbf{h}}_{i-1} + \overrightarrow{\mathbf{z}}_i \odot \overrightarrow{\mathbf{h}}'_i & \text{if } i > 0 \\ 0 & \text{if } i = 0, \end{cases} \tag{5.2}$$

where the vectors are given by

$$\overrightarrow{\mathbf{h}}'_i = \tanh\left(\overrightarrow{\mathbf{W}}\mathbf{E}'\mathbf{x}_i + \overrightarrow{\mathbf{U}}\left[\overrightarrow{\mathbf{r}}_i \odot \overrightarrow{\mathbf{h}}_{i-1}\right]\right) \tag{5.3}$$

$$\overrightarrow{\mathbf{z}}'_i = \sigma\left(\overrightarrow{\mathbf{W}}_z\mathbf{E}'\mathbf{x}_i + \overrightarrow{\mathbf{U}}_z\overrightarrow{\mathbf{h}}_{i-1}\right) \tag{5.4}$$

$$\overrightarrow{\mathbf{r}}'_i = \sigma\left(\overrightarrow{\mathbf{W}}_r\mathbf{E}'\mathbf{x}_i + \overrightarrow{\mathbf{U}}_r\overrightarrow{\mathbf{h}}_{i-1}\right), \tag{5.5}$$

where $\mathbf{E}' \in \mathbb{R}^{m \times \Sigma_{src}}$ is the symbol embedding matrix. We have that $\overrightarrow{\mathbf{W}}, \overrightarrow{\mathbf{W}}_z, \overrightarrow{\mathbf{W}}_r \in \mathbb{R}^{n \times m}$ and $\overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{U}}_z, \overrightarrow{\mathbf{U}}_r \in \mathbb{R}^{n \times n}$ are weight matrices. Moreover, m is the symbol embedding dimensionality, n is the number of hidden units, and σ represents the sigmoid function (section 2.4.2).

The encoder's backward states $\left(\overleftarrow{\mathbf{h}}_1 \dots \overleftarrow{\mathbf{h}}_{T_x}\right)$ are computed similarly to the above forward states. It is important to note that the forward and backward RNN share word embeddings E', but not weight matrices. The final encoded state sequence is a concatenation between the forward and backwards pass, such that

$$\mathbf{h}_i = \left[\overrightarrow{\mathbf{h}}_i^\top \overleftarrow{\mathbf{h}}_i^\top\right]. \tag{5.6}$$

5.2.2 The `char2word` Encoder

The `char2word` encoder samples states from the forward pass of the `char` encoder defined in the above section. The states it samples are based on the appearance of spaces in the original text, leaving a word style sampling with character encoded embeddings. We can consider these as word embeddings. Given our `grid_gather()` operation (see section 4.8.3) we sample the indices from $\overrightarrow{\mathbf{h}}$, such that

$$\mathbf{h}_i^{\text{spaces}} = \overrightarrow{\mathbf{h}}_{\varphi_i}, \tag{5.7}$$

where $\overrightarrow{\mathbf{h}}_i$ is defined from above section at eq. (5.2) and φ is an ordered list of indices gathered from our `grid_gather()` function on the input matrix $\mathbf{X}$. Thus, $\mathbf{h}_i^{\text{spaces}}$ gives the i$^{\text{th}}$ space in the input sequence. Given $\mathbf{h}^{\text{spaces}}$ the formulas from the bi-directional char encoder is used, with $\mathbf{h}_i^{\text{spaces}}$ replacing $\mathbf{E}'\mathbf{x}_i$.

5.2.3 The `char` Decoder

For our decoder we use an RNN with GRUs [Cho et al., 2014c] as introduced in sections 2.7, 2.7.2 and 2.7.4 with an attention mechanism [Bahdanau et al., 2014] as introduced in section 2.7.4. The decoder works on a character basis and predicts one character at a time. We will omit bias terms whenever possible to make the equations less cluttered.

The new state $\mathbf{s}_i$ using n hidden units is computed by

$$\mathbf{s}_i = \begin{cases} f(\mathbf{s}_{i-1}, \mathbf{y}_{i-1}, \mathbf{c}_i) = (1 - \mathbf{z}_i) \odot \mathbf{s}_{i-1} + \mathbf{z}_i \odot \mathbf{s}' & \text{if } i > 0 \\ \mathbf{h}_{T_x} & \text{if } i = 0, \end{cases} \tag{5.8}$$

where $\odot$ represents element-wise multiplication and $\mathbf{z}_i$ is the output computed by the update gate (explained below). The proposed update state $\mathbf{s}'$ is computed by

$$\mathbf{s}'_i = \tanh(\mathbf{W}e(\mathbf{y}_{i-1}) + \mathbf{U}[\mathbf{r}_i \odot \mathbf{s}_{i-1}] + \mathbf{C}\mathbf{c}_i), \tag{5.9}$$

where $e(\mathbf{y}_{i-1})$ is an m-dimensional embedding (see section 2.8) of the symbol $\mathbf{y}_{i-1}$, in our case a character. $\mathbf{r}_i$ is the output of the reset gate (elaborated below).

The update gate $\mathbf{z}_i$ allow the hidden units of the RNN to maintain previous knowledge (activations), while the reset gate $\mathbf{r}_i$ controls how much of previous states should be forgotten. They are computed by

$$\mathbf{z}_i = \sigma(\mathbf{W}_z e(\mathbf{y}_{i-1}) + \mathbf{U}_z \mathbf{s}_{i-1} + \mathbf{C}_z \mathbf{c}_i), \tag{5.10}$$

$$\mathbf{r}_i = \sigma(\mathbf{W}_r e(\mathbf{y}_{i-1}) + \mathbf{U}_r \mathbf{s}_{i-1} + \mathbf{C}_r \mathbf{c}_i), \tag{5.11}$$

where σ represents the sigmoid function. To compute the output we use a linear projection of $\mathbf{s}_i$ and a softmax to normalise the linear projection into a probability space. This is computed by

$$p(\mathbf{y}_i |\, \mathbf{s}_i, \mathbf{y}_{i-1}, \mathbf{c}_i) = \frac{\exp\left(\mathbf{W}_y \mathbf{s}_i\right)}{\sum_{k=1}^{T_x} \exp\left(\left(\mathbf{W}_y \mathbf{s}_i\right)_k\right).} \tag{5.12}$$

5.2.4 Alignment Model

The alignment model a (defined in eq. (2.70)) is used to compute the context c_i for time step i, which is utilised by the decoder to perform variable length attention. To create our alignment function, a, we use the following formula

$$a(\mathbf{s}_{i-1}, \mathbf{h}_j) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a \mathbf{s}_{i-1} + \mathbf{U}_a \mathbf{h}_j), \tag{5.13}$$

where $\mathbf{W}_a \in \mathbb{R}^{n \times n}$, $\mathbf{U}_a \in \mathbb{R}^{n \times 2n}$ and $\mathbf{v}_a \in \mathbb{R}^n$. As $\mathbf{U}_a \mathbf{h}_j$ does not depend on i, we can pre-compute it in advance for optimisation purposes.

5.3 Training Procedure

Responsible: Elias Obeid (s142952)

This section presents how we trained our models. Moreover, we discuss how data flows as input to produce a prediction (section 5.3.1). We dig into how a given batch is used in our models in section 5.3.1.1. For clarification, in the following we will refer to `char2word` and `char` encoders. These will be illustrated with figures and are represented with encapsulated dotted red boxes. Both of these are connected to a character-level decoder and the models are thus respectively referred to as `char2word-to-char` and `char-to-char` models.

While training our models we used the optimiser called Adam [Kingma and Ba, 2014]. We presented the Adam optimiser in section 2.6.2.4 along with other optimisation algorithms (sections 2.6.2.1 to 2.6.2.3). We introduce our loss function in section 2.9.1. Adam utilises historical information from its past executions and the amount of historical information used can be determined by the hyperparameters β_1 and β_2. We use the default settings by [Kingma and Ba, 2014], i.e.

$$\beta_1 = 0.9,\ \beta_2 = 0.999,\ \alpha = 0.001, \tag{5.14}$$

where α is the learning rate.

We found that these settings minimised the training loss best. Moreover, to stabilize the training and avoid exploding weights, we applied L2 regularisation, as described in section 2.6.3, with a small value of lambda given by $\lambda = 0.000{,}001$. To avoid exploding gradients we normalised the gradients, as described in section 2.7.3, if it exceeded a norm of 1. We present the models' parameters along with illustrations for the `char-to-char` and `char2word-to-char` models in figs. 5.4 and 5.7 and tables 5.5 and 5.8. For these models we defined the number of classes to be 300, i.e $\Sigma_{src} = \Sigma_{trg} = 300$.

5.3.1 From Input to Target

The figs. 5.4 and 5.7 crudely present how data flows between the different layers. The first layer (the one at the bottom) receives the input and feeds it to the next layer, which computes the character embeddings w.r.t. the given sequence. The data flow is the same for these two layers for both models.

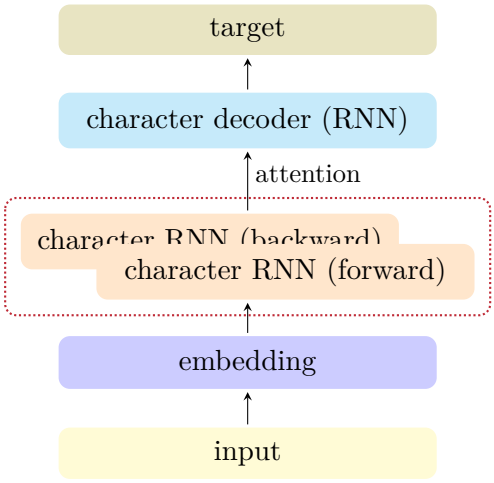

Figure 5.4: model architecture.

layer	no. units
input ($\mathbf{X}$)	$\Sigma_{src} = 300$
embedding	256
char RNN (forward)	400
char RNN (backward)	400
attention	300
char decoder	400
target ($\mathbf{T}$)	$\Sigma_{trg} = 300$

Table 5.5: model parameters.

Figure 5.6: `char-to-char` visualisation and specifications, where Σ_{src} and Σ_{trg} represent the number of classes in the source and target languages, respectively. The red dotted box encapsulating the two character RNNs constitute the `char` encoder.

Specifics of `char-to-char`

For the pure character-level model the embeddings are feed to the character RNNs which perform transformations for both forward and backward sequences. These are encapsulated with a red dotted box in fig. 5.4 and together constitute the `char` encoder. The character decoder now receives the transformations from both RNNs along with an alignment model (the attention). The decoder can then choose which characters to focus on when translation each element.

Specifics of `char2word-to-char`

The `char2word-to-char` model feeds the embeddings to a single forward character RNN to perform transformations. From that character RNN the model gather the words with our `grid_gather()` method (section 4.8.3) followed by the spaces RNNS, which construct the transformed words backwards and forward. These three RNNs constitute the `char2word` encoder, which is illustrated in fig. 5.7 with a red dotted box. Similar to the `char-to-char` model, the character decoder receives the transformations from the two spaces RNNs with an attention model.

Note, that while training the character decoder receives the previous time steps correct prediction for the next time step. The decoder then finally concludes with a prediction and it is compared to the expected target. The loss function thus dictates the parameter updates carried out by the Adam optimiser and the Backpropagation algorithm (section 2.6.1).

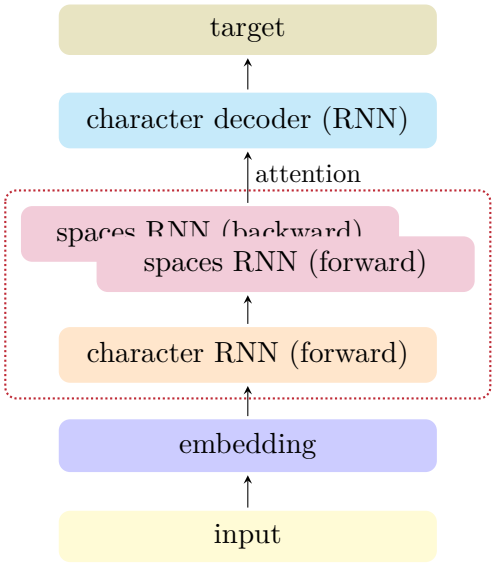

Figure 5.7: model architecture.

layer	no. units
input (**X**)	$\Sigma_{src} = 300$
embedding	256
char RNN (forward)	400
spaces RNN (forward)	400
spaces RNN (backward)	400
attention	300
char decoder	400
target (**T**)	$\Sigma_{trg} = 300$

Table 5.8: model parameters.

Figure 5.9: `char2word-to-char` visualisation and specifications, where Σ_{src} and Σ_{trg} represent the number of classes in the source and target languages, respectively. The red dotted box encapsulating the character RNN and the two spaces RNNs constitute the `char2word` encoder.

5.3.1.1 Walking Through Batch Usage

In section 4.4 we introduced what batches consist of. Here we will describe in more detail how a batch is used in the models.

The model receives the `x_encoded` list as input together with the `x_len` indicating the size of the respective samples. The very first layer creates character embeddings. The character-level RNNs read individual character embeddings and produce some output. For the `char2word-to-char` model this output is used to construct output that may be considered as word embeddings. The `grid_gather()` method uses these embeddings together with the `x_spaces` list to fetch the respective words/tokens defined by the input sample.

The `char2word` encoder outputs some representation of the words along with an alignment model for attention. The RNNs (constituting the `char2word` encoder) are invoked on the original sequence and the reverse of the sequence to produce a combined representation of both forward and backwards sequence.

The model's decoder uses the `char2word` encoder's output representation and state to produce a prediction. It tries to reproduce the sequence given by the `t_encoded_go` list. The loss is finally computed using the sequences in `t_encoded`. While computing the loss the `t_mask` is used to only compute the part of the sequence which we are interested

in. This is done because of dynamic sequence lengths.

The word embeddings produced in the `char2word` encoder reduce the amount of data which must be paid attention to because of focusing on single words (and not individual characters). Moreover, they reduce the model's memory requirements because one word is easier to remember than many characters.

5.4 Results

Responsible: Alexander Johansen (s145706)

This section presents the results we obtained. We present quantitative results in section 5.4.1 and qualitative results in section 5.4.2.

5.4.1 Quantitative Results

This section presents our results on the Europarl dataset (table 5.10) and our results on the WMT'15 dataset (table 5.11). Our models were trained for five days (160,000 iterations) and validated every 1000 iteration. The *statmt* results are from their evaluation matrix at `http://matrix.statmt.org/matrix`.

Table 5.10 presents the translation performances (BLEU scores) of our experiments with the Europarl datasets. We present results on different language pairs as defined by the *language* column in the tables. This column gives the from and to language as source–target. These languages are Danish (Da), English (En), German (De), and French (Fr).

		validation set	*test sets*		
Model	Language	devtest2016	test2006	test2007	test2008
statmt	Da–En	29.40	30.00	30.03	30.02
`char2word-to-char`	Da–En	25.73	25.01	25.19	25.89
statmt	De–En	27.00	26.70	27.70	28.30
`char2word-to-char`	De–En	22.70	22.11	21.45	22.76
statmt	En–De	19.90	19.90	20.40	20.60
`char2word-to-char`	En–De	17.61	16.37	17.05	17.56
statmt	Fr–En	30.30	30.10	30.60	32.30
`char2word-to-char`	Fr–En	25.65	24.87	25.84	25.97

Table 5.10: Results: Europarl V7, *devtest2006* was used as validation set, while *test2006*, *test2007*, and *test2008* were used as test sets.

The results from the WMT'15 dataset are presented in table 5.11. Chung et al. [2016], Luong et al. [2015b] provide both single model and ensemble (the results in parenthesis), showing that we are 3-5 BLEU away from state-of-the-art on our single model performance. However, we removed the ensemble results given by Luong et al. [2015b] because

we found that reporting unknown-replacement would not give comparable results. This meant that they reported a correct prediction on unknown words/symbols, which is misleading in this context.

Model	Language	*validation set* newstest2013	*test sets* newstest2014	newstest2015
statmt, Bojar et al. [2013, 2014, 2015]	De–En	28.4	29.0	29.3
`char2word-to-char`	De–En	20.15	19.03	19.90
statmt, Bojar et al. [2013, 2014, 2015]	En–De	20.2	20.60	24.9
Luong et al. [2015b]	En–De	-	19.00 (*)	22.80 (*)
Chung et al. [2016]	En–De	21.57 (23.14)	21.33 (23.11)	23.45 (25.24)
`char2word-to-char`	En–De	16.78	15.04	17.43

Table 5.11: Results: WMT'15, *newstest2013* was used as validation set, *newstest2014* and *newstest2015* were used as test sets. The results in parenthesis are from running an ensemble of models.

5.4.1.1 Complications

After performing our final experiments for the `char2word-to-char` and `char-to-char` models, we made a mistake and defined our `char-to-char` configuration files to point to the `char2word-to-char` code base. This meant that we ran duplicate models of `char2word-to-char` and no `char-to-char` models. Furthermore, we found a complication in the default TensorBoard setting, which only saved the last five checkpoints corresponding to the last 5000 iterations. We had not altered these settings and often the best performing iteration was not among these last iterations. Some of our models had up to 1 BLEU better performance on validation, not in the last 5000. On average our models are about 4-5 BLEU lower than what has been published in ACL on the official WMT and Europarl datasets. However, we could imagine that an ensemble would put us much closer to those results.

5.4.1.2 Length Analysis

To understand how effective the attention is on sentences of different lengths, and the effect of translating sentences that are longer than those that the model was trained on, we performed a length analysis presented in fig. 5.12. The length analysis is based on the *newstest2015* dataset from WMT'15 En–De. Such that all data points within a given interval, i.e. sequence length 100 to 125, have been used to perform the validation. Figure 5.13 shows the distribution of sample lengths within the newstest '15 dataset.

We only trained our model with source sequence length of up to 250 (by following Chung et al. [2016] preprocessing guidelines). However, for validation we test the model with sequence lengths of up to the maximum sequence length of 500. Sequence lengths between 250 to 500 represents approx. 4.5% of the total dataset. Further, to compare results we found that Luong et al. [2015b] has a length analysis of a word-to-word model

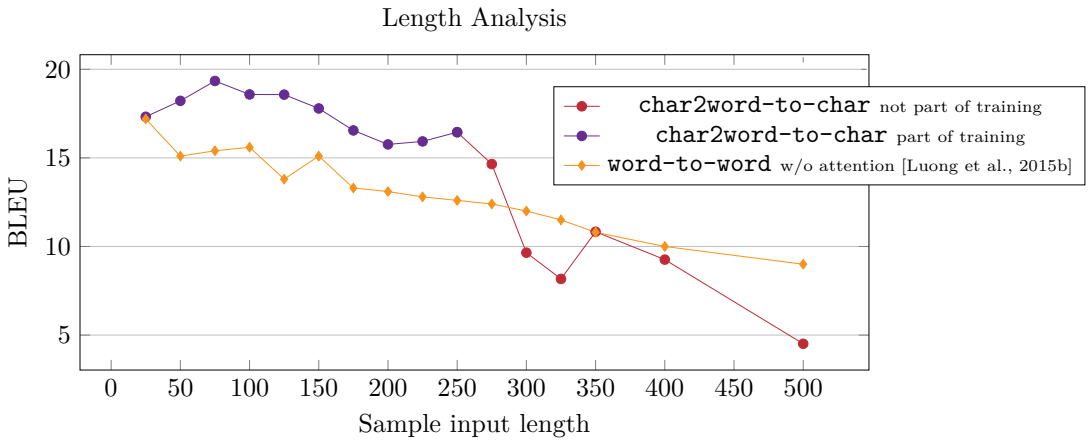

Figure 5.12: Bleu scores on test samples (from newstest '15) of different lengths. Since our model (char2word-to-char) was only trained on input sequences up to length 250, the red points indicate test scores on samples that are longer than what the model was trained on. The blue points indicate test scores on samples that are within the range of sequence lengths that the model was trained on.

without attention on the WMT'15 En-De *newstest2015*, which we have plotted alongside our model.

5.4.2 Qualitative Results

This sections presents more qualitative results with illustrations to emphasis our results.

5.4.2.1 Alignment

The attention used in our `char2word-to-char` model provides an intuitive way to inspect how the model learns to represent variable length sequences. This is done by visualising the annotated weights α_{ij} from eq. (2.69) as illustrated in figs. 5.14 to 5.19 where we randomly sampled a short sentence for each of the Europarl models and a long sentence for each for the WMT'15 models. Each figure has a matrix with the attention annotations from each time step when the model predicts the next character. Such that the most prominent activations are illustrated with a dark blue color, which illustrates how the decoder utilises the source sentence to produce the target sentence. The x-axis and y-axis of each plot corresponds to the characters in the target sequence and words in the source sequence.

From the figures there is a clear dependency between words in the source sequence and characters being translated in the target sequence. Interestingly, the character decoder often only places attention on the source word when decoding the first few symbols of the target word, such as when writing `requirement` in fig. 5.17 or `Institution` in fig. 5.15. The remaining attention is either used on finding the next word, such as `now`

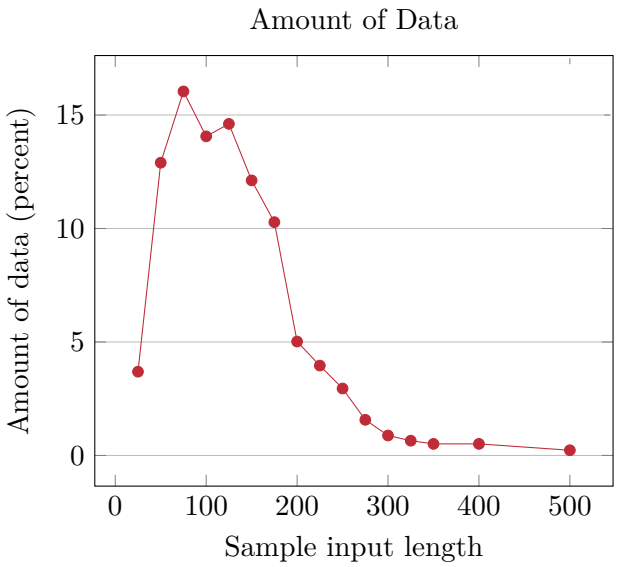

Figure 5.13: The percentage of samples in the newstest '15 dataset that are within each range of lengths.

in fig. 5.16 (Europarl Da–En), looking to the end-of-sequence `<EOS>` symbol (we assume to either to get a summarised view of the sentence or to consider finishing it), such as `proposal` in fig. 5.14 (Europarl De–En) or simply not use the attention function at all by taking an alignment that looks close to uniform, such as with `Institutionen` in fig. 5.15 (Europarl En–De).

The character decoder's use of soft-alignment is not common in word-to-word based models such as illustrated in Bahdanau et al. [2014], but BPE-to-character based models (character decoder) illustrated by Chung et al. [2016] seem to work similarly to ours. Further, the attention seems to work properly on larger sentences, which is visualised in fig. 5.18 (WMT De–En) and fig. 5.19 (WMT En–De).

The presented plots have been created with the script given in appendix B.2.

5.4.2.2 Sample Translations

In tables 5.20 and 5.21 we show sample translations from the test set (*test2008* for Europarl and *newstest2015* for WMT'15). Where *src* refers to source language, *ref* refers to reference on the target language, *gt* refers to a translation by google translate [8] and *ours* refers to the translation from our *char2word-to-char* model.

It is interesting to notice how the char based decoder is able to correctly translate names of persons and places, such as *Moreira Da Silva* in the **Danish-English** translation from Europarl. Further, in the **German-English** translation on Europarl, we illustrate how

[8] `https://translate.google.com/`

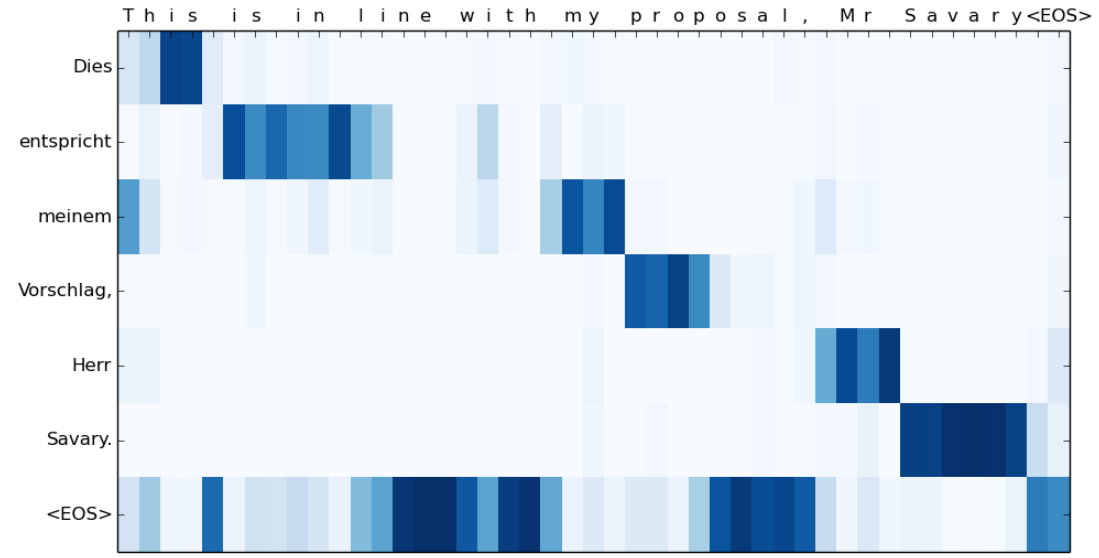

Figure 5.14: Attention matrix for German to English (De–En) – Europarl, devtest2006.

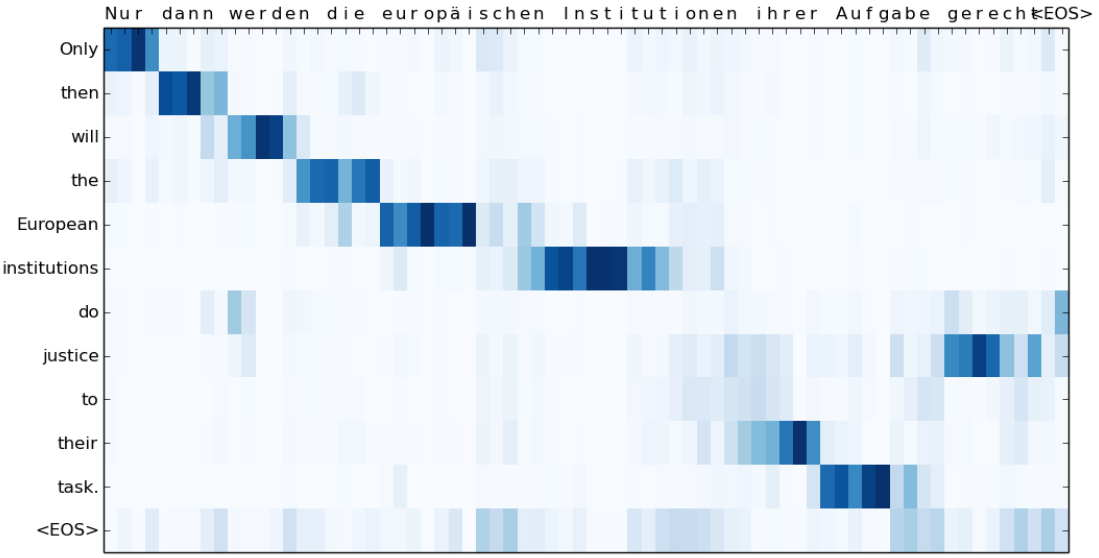

Figure 5.15: Attention matrix for English to German (En–De) – Europarl, devtest2006.

our algorithm occasionally can go into loops copying previous states, and ending up with a poor translation.

It is to note that BLEU (section 2.9.2) default implementation by moses is meant to be evaluated on a corpus, as it will leave the sentence at zero BLEU if no 4-gram had been found. Because of such, we also supply the 1-/2-/3-/4-grams found.

In table 5.21 based on WMT'15, we supply samples of long sentence translations.

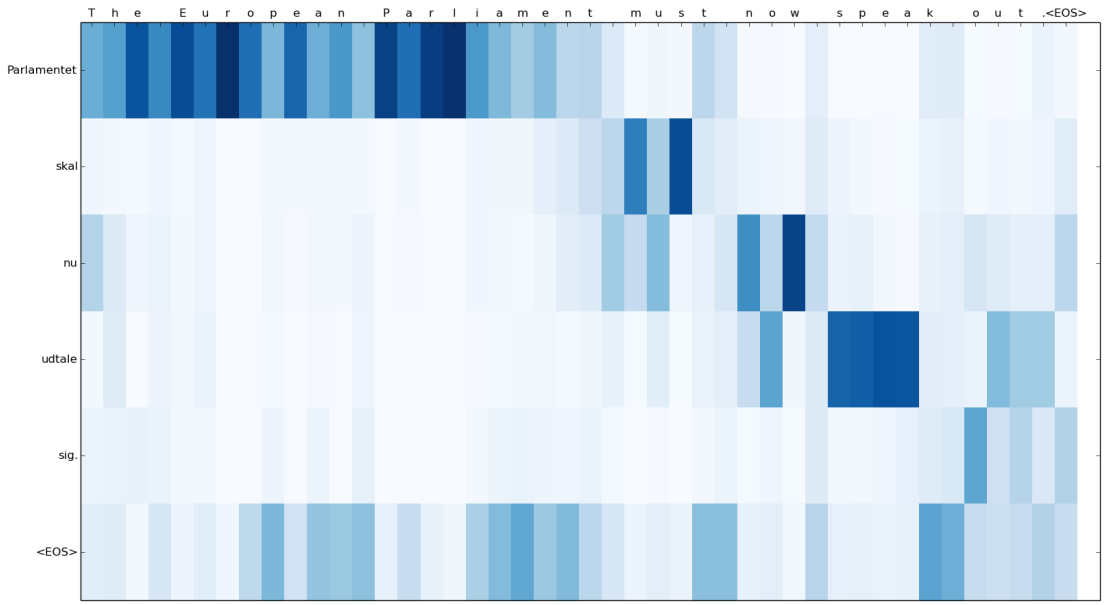

Figure 5.16: Attention matrix for Danish to English (Da–En) – Europarl, devtest2006.

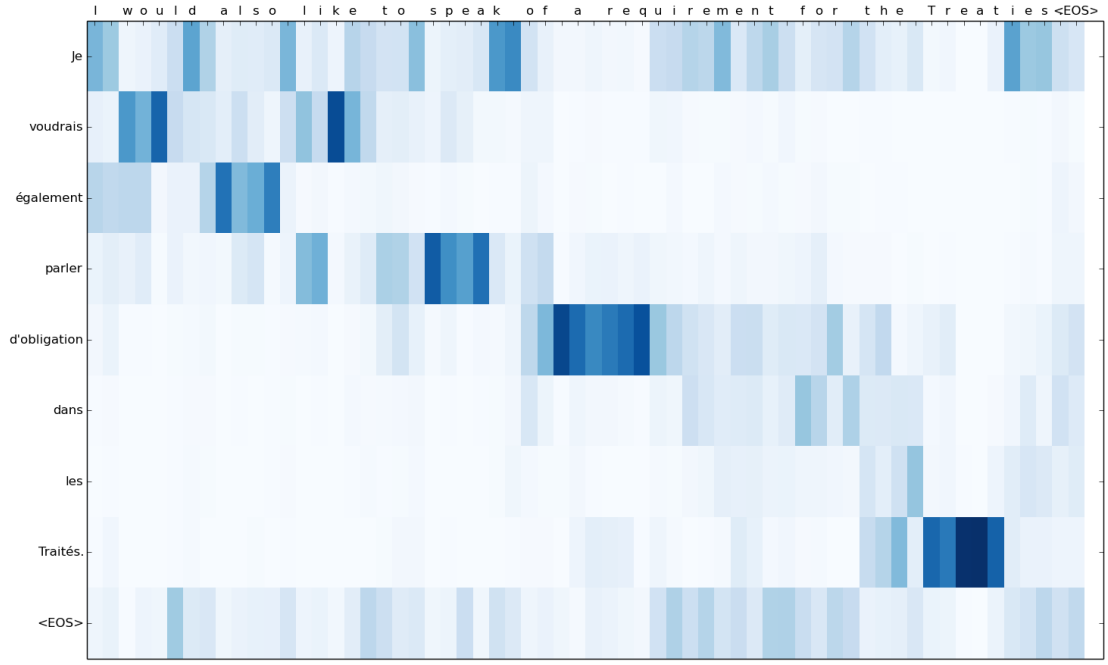

Figure 5.17: Attention matrix for French to English (Fr–En) – Europarl, devtest2006.

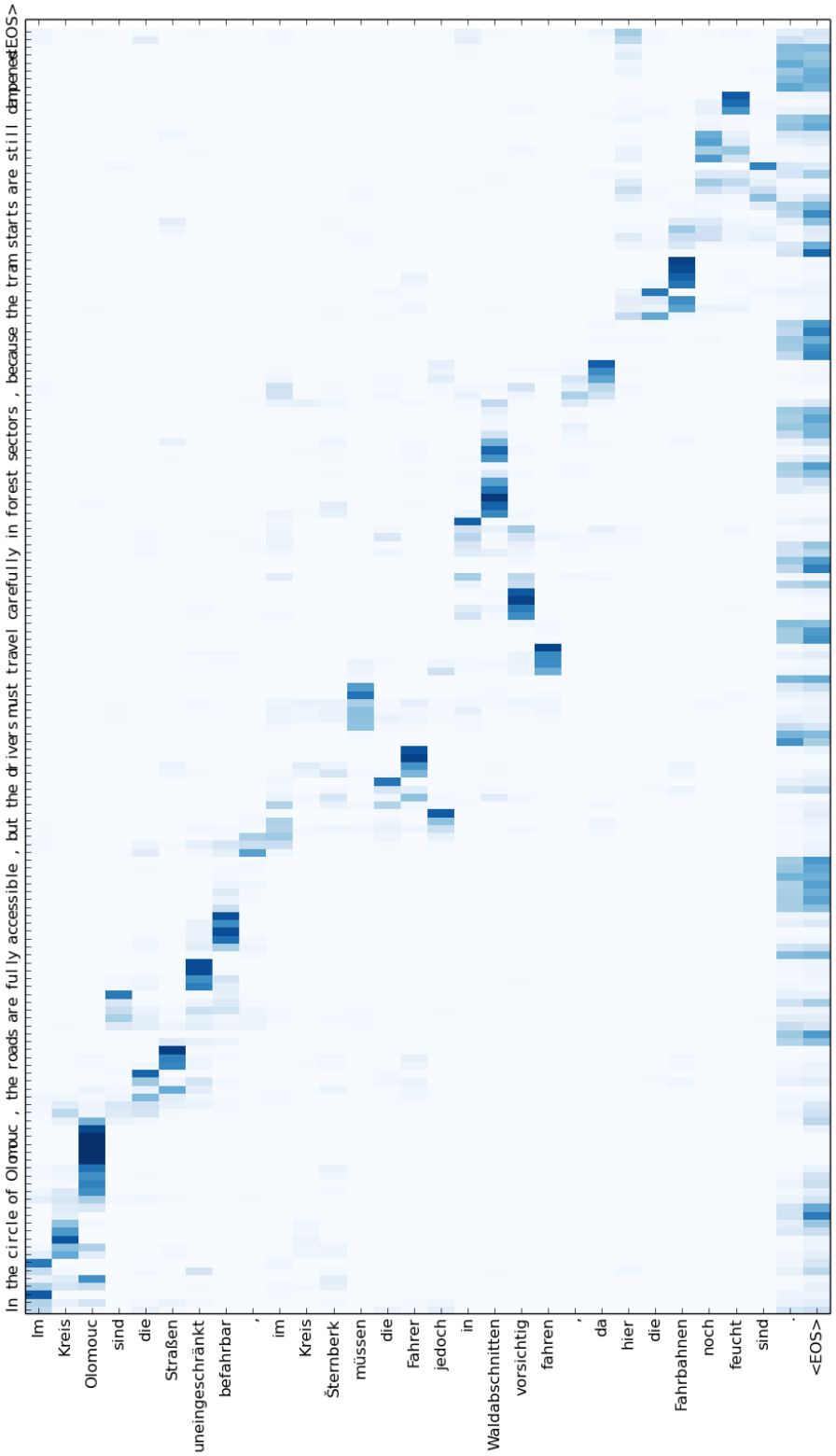

Figure 5.18: Attention matrix for German to English (De–En) – WMT'15, newstest2013.

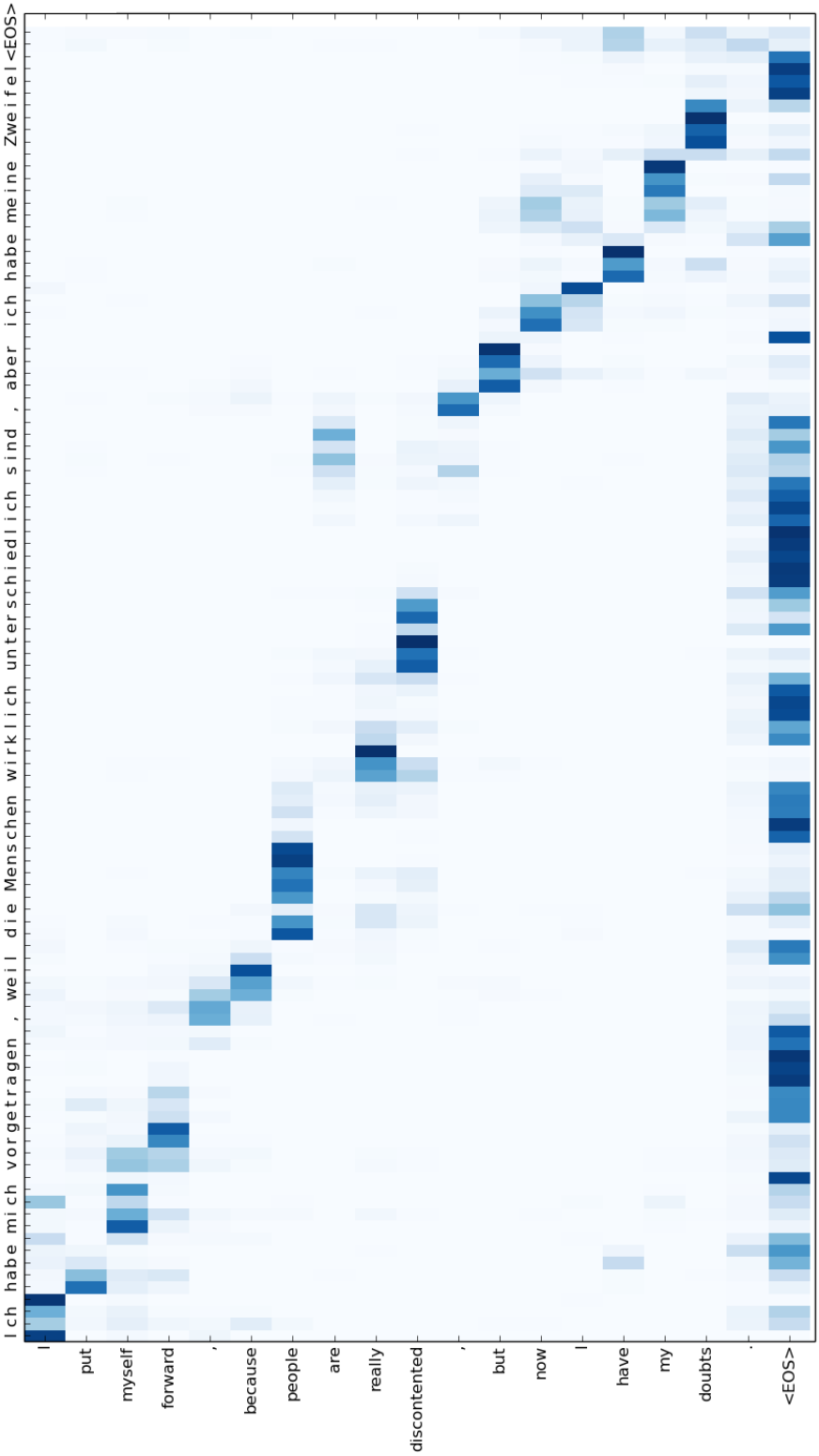

Figure 5.19: Attention matrix for English to German (En–De) – WMT'15, newstest2013.

Table 5.20: Sample translations from Europarl on *test2006*.

Danish-English	
src	Betænkning af Moreira Da Silva (A5-0271/2000)
ref	Moreira Da Silva Report (A5-0271/2000)
gt	Report: Moreira Da Silva (A5-0271 / 2000) **BLEU = 0.00 (42.9/33.3/20.0/0.0)**
ours	- Moreira Da Silva report (A5-0271/2000): **BLEU = 0.00 (50.0/40.0/25.0/0.0)**
German-English	
src	Die Liberalen vertreten den Grundsatz "so wenig Staat wie möglich ", wobei die USA als Beispiel angeführt werden.
ref	Liberals tell us that as little State involvement as possible is required.
gt	The Liberals represented the principle "as little government as possible", said USA be cited as an example. **BLEU = 0.00 (23.5/0.0/0.0/0.0)**
ours	The Liberals represent the principle 'to 'to 'to 'to 'to 'to 'to 'to 'to 'to 'the 'United States' as examples of the United States are listed as an example. **BLEU = 0.00 (10.3/0.0/0.0/0.0)**
English-German	
src	I have studied carefully what the Commissioner said to my colleague Mr Hume: "No massive job losses in this sector.
ref	Ich habe mir ganz genau angehört, was das Kommissionsmitglied zu Herrn Hume sagte: "Keine massiven Arbeitsplatzverluste in diesem Sektor.
gt	Ich habe sorgfältig untersucht , was der Kommissar sagte zu meinen Kollegen, Herrn Hume: "Keine massiven Verlust von Arbeitsplätzen in diesem Sektor. **BLEU = 0.00 (45.5/19.0/5.0/0.0)**
ours	Ich habe mich genau um das geäußert, was der Kommissar Herrn Hume hier gesagt hat: "Nicht massive Arbeitsplätze in diesem Sektor. **BLEU = 0.00 (47.6/20.0/5.3/0.0)**
French-English	
src	Je rappelle les priorités principales assignées au programme CARDS : le renforcement institutionnel dans le sens de la démocratie et de l'État de droit, le développement économique et la reconstruction, la coopération régionale.
ref	I should like to remind you that the main priorities of the CARDS programme are the strengthening of institutions with regard to democracy and the rule of law, economic development and reconstruction, and regional cooperation.
gt	I recall the main priorities assigned to the CARDS program: institutional strengthening in the sense of democracy and the rule of law, economic development and reconstruction, regional cooperation. **BLEU = 34.08 (78.6/48.1/34.6/28.0)**
ours	I would like to recall the main priorities assigned to the CARDS programme: the institutional strengthening of democracy and the rule of law, economic development and reconstruction, regional cooperation. **BLEU = 36.31 (82.8/53.6/33.3/26.9)**

Table 5.21: Sample translations from WMT'15 on *newstest2015*

German-English	
src	Eine vergleichbare Studie der US-Handelskammer zu Beginn dieses Jahres zeigte , dass in Virginia geringere staatliche und örtliche Unternehmenssteuern verlangt werden und insgesamt ein besseres Steuerklima herrscht als in Maryland .
ref	A similar study by the U.S. Chamber of Commerce earlier this year showed that Virginia had lower state and local business taxes and an overall better business tax climate than Maryland.
gt	A similar study by the US Chamber of Commerce earlier this year showed that in Virginia less state and local business taxes are required and a better overall tax climate prevails as in Maryland. **BLEU = 43.94 (70.6/48.5/37.5/29.0)**
ours	A comparable study of the US Chamber showed that in Virginia , less state and local companies were required and a better tax climate than in Maryland . **BLEU = 0.00 (64.3/22.2/7.7/0.0)**
English-German	
src	Norwegian Cruise Line Holdings Ltd NCHL.O , the world 's third largest cruise operator , is in advanced talks to acquire peer Prestige Cruises International Inc for around $ 3 billion , according to people familiar with the matter .
ref	Norwegian Cruise Line Holdings Ltd NCHL.O , der drittgrößte Kreuzfahranbieter der Welt , führt laut Kennern der Branche fortgeschrittene Verhandlungen zum Erwerb des Konkurrenten Prestige Cruises International Inc für etwa 3 Milliarden $.
gt	Norwegian Cruise Line Holdings Ltd NCHL.O, der weltweit drittgrößte Reederei, ist in fortgeschrittenen Gesprächen Peer Prestige Cruises International Inc für rund 3 $ zu erwerben Milliarden, nach Menschen mit der Angelegenheit vertraute. **BLEU = 22.58 (46.9/25.8/20.0/13.8)**
ours	Norwegische Kreuzfahrtlinien Ltd. Ltd. NCLL.Ho , der drittgrößte Kreuzfahrer der Welt , liegt in den fortgeschrittenen Gesprächen , um die Peer Prestige Cruises Inter auf etwa 3 Milliarden Dollar zu bekommen , gemäß den Menschen , die mit der Angelegenheit vertraut sind . **BLEU = 0.00 (30.2/16.7/7.3/0.0)**

CHAPTER 6

Conclusion

The conventional approach to neural machine translation, called encoder-decoder attention approach, uses word level encodings of language pairs to translate sentences. However, this is problematic as using word level encodings requires large vocabularies, being expensive, often needs to use unknown symbols for names and places and not all languages have word lexemes.

In this thesis we address this issue by using character level encodings instead of word level encodings. However, using character level encodings our sentences becomes about 5 times longer, which is a more challenging task for the recurrent neural networks applied in the sequence-to-sequence model as well as more computationally intensive

We propose using a novel encoder-decoder attention approach on character level encodings using a customised data loader to speed up training. Further we test a hierarchical model by gathering indices in our character level encoder based on the occurrence of certain lexemes (spaces) in the input sentence.

Our novel encoder-decoder on character level encodings, and our hierarchical model performed similarly. Our results show that our single model performance is slightly worse than state-of-the-art single model performance for word level translation. Our `char2word-to-char` model performed with the following test results on the Europarl dataset; Danish to English of 25.89 BLEU, German to English of 22.76 BLEU, English to German of 17.56 BLEU, and French to English of 25.97 BLEU. Moreover, the model performed with the following test results on the WMT'15 dataset; German to English 19.90 BLEU, and English to German 17.43 BLEU.

Based on the results presented in section 5.4 we conclude that our character level model is able to perform nearly as well as state-of-the-art models. It does not suffer from unknown tokens, and it does not have any trouble spelling words or any grammatical challenges. Further, we conclude that our model solves the problem statement presented in section 1.4 and translates many sentences in an understandable and natural manner. Overall our model performs rather well.

6.1 Future Work

We would like to investigate hierarchical models for the encoder and decoder. For the encoder Cho et al. [2014a] proposes to use a Gated Recursive Convolutional Neural Network (grConv) which weaves the encoded sequence into a binary hierarchical tree. However, Cho et al. [2014a] does not propose a method to perform attention on his

grConv architecture. It could be interesting to experiment with methods for building an attention mechanism for recursively exploring the hierarchical layers of such binary hierarchical tree, especially with a GRU/LSTM, as this would allow extracting information at the most suitable level of encoding. Further using a sparse softmax [Martins and Astudillo, 2016] would allow sparsely interacting with each layer, making it possible to do a binary search instead of a linear search when applying attention on a hierarchical encoder.

A hierarchical decoder may make it easier for the neural network to remember what is has previously written, as well as structure larger corpora of text. We believe a dual network (one to control what has been said and one to control what to say next) with a slight modification of the grConv structure could do such.

Moreover, we could consider optimising our RNN by stabilising activations [Krueger and Memisevic, 2015], optimise our architecture [Zhang et al., 2016], applying batch normalisation [Cooijmans et al., 2016] or solving memory issues [Gruslys et al., 2016] so we can try building a deep ResNet [van den Oord et al., 2016].

6.2 Acknowledgements

We would like to thank our supervisors for this project, Ole Winther and Casper Kaae Sønderby, for their helpful insights during this thesis. Further, we would like to thank The Technical University of Denmark's Compute Department for granting access to their server equipment throughout our thesis.

Side Note

If the reader finds any part of this report to be of poor quality in any way we blame the release of the Pokémon Go game. We do apologise for this — we are but humans.[1]

[1] This is a joke – thank you for reading.

APPENDIX A

Partial Derivatives of Mean Squared Error for Linear Regression

In section 2.2 we presented the linear regression algorithm. This appendix gives a walk-through of deriving the partial derivatives of the mean squared error loss function.

Partial derivatives are calculated as regular derivatives of the given function, except that the all but the variable of interest are held fixed while differentiation.

- The derivative of $f(x) = cx^n$ is $f'(x) = cnx^{n-1}$ (power rule)
- The derivative of a constant is 0 (constant rule)
- Summations do not affect the derivative, and are simply copied down

Furthermore, the chain rule says that for chained functions, e.g. $g\left(f(x)\right)$, we treat $f(x)$ as the variable while calculating the derivative of $g\left(f(w)\right)$ and multiply by the derivative of $f(x)$, i.e. $\frac{d}{dx} = g'\left(f(x)\right) \times f'(x)$. If we defined $g(x)$ as our cost function from eq. (2.2) we would have

$$g\left(f(\mathbf{w})^{(i)}\right) = \frac{1}{2m}\sum_{i=1}^{m}\left(\mathbf{w}^\top \mathbf{X}_i - \mathbf{t}_i\right)^2 \tag{A.1}$$

and splitting the functions gives us

$$g(\mathbf{w}) = \frac{1}{2m}\sum_{i=1}^{m}\left(f(\mathbf{w})^{(i)}\right)^2 \tag{A.2}$$

$$f(\mathbf{w})^{(i)} = \mathbf{w}^\top \mathbf{X}_i - \mathbf{t}_i \tag{A.3}$$

The following will walk through calculating the partial derivatives to the weights in the functions. We assume that $\mathbf{w}$ has two values, i.e. $\mathbf{w}^\top = (w_0, w_1)$.

Now, calculating the partial derivative w.r.t. the first weight, w_0, we treat the function

$f(\mathbf{w})^{(i)}$ as a variable and first calculate the partial derivative for $g\left(f(\mathbf{w})^{(i)}\right)$ as

$$\frac{\partial}{\partial w_0} g(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} \left(f(\mathbf{w})^{(i)}\right)^2 \tag{A.4}$$

$$= 2 \times \frac{1}{2m} \sum_{i=1}^{m} \left(f(\mathbf{w})^{(i)}\right)^{2-1} \frac{\partial}{\partial w_0} w_0 \tag{A.5}$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(f(\mathbf{w})^{(i)}\right) \times 1 \tag{A.6}$$

Next, we calculate the partial derivative w.r.t. w_0 for $f(\mathbf{w})^{(i)}$ as

$$\frac{\partial}{\partial w_0} f(\mathbf{w})^{(i)} = \frac{\partial}{\partial w_0} \left(w_0 X_{i,0} + w_1 X_{i,1} - t_i\right) \tag{A.7}$$

$$= \frac{\partial}{\partial w_0} w_0 \tag{A.8}$$

$$= 1, \tag{A.9}$$

because we have that $X_{i,0} = 1$ and that $w_1, \mathbf{X}_{i,1}$, and $\mathbf{t}_i$ are constants when calculating $\frac{\partial}{\partial w_0}$. We have that $\mathbf{X}_{i,0}$ gives the first element of the i^{th} input vector. To finish and calculate the final result we apply the chain rule and combine the two above derivatives and get

$$\frac{\partial}{\partial w_0} g\left(f(\mathbf{w})^{(i)}\right) = \frac{\partial}{\partial w_0} g(w) \frac{\partial}{\partial w_0} f(\mathbf{w})^{(i)} \tag{A.10}$$

$$= \frac{1}{m} \sum_{i=1}^{m} f(\mathbf{w})^{(i)} \frac{\partial}{\partial w_0} f(\mathbf{w})^{(i)} \tag{A.11}$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(w_0 + w_1 \mathbf{X}_{i,1} - \mathbf{t}_i\right) \times 1 \tag{A.12}$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(w_0 + w_1 \mathbf{X}_{i,1} - \mathbf{t}_i\right) \tag{A.13}$$

To calculate partial derivative w.r.t. to the next weight, w_1, we need to calculate $f(\mathbf{w})^{(i)}$ because the term for $g\left(f(\mathbf{w})^{(i)}\right)$ will not change. Thus, we have the following

$$\frac{\partial}{\partial w_1} f(\mathbf{w})^{(i)} = \frac{\partial}{\partial w_1} \left(w_0 + w_1 \mathbf{X}_{i,1} - \mathbf{t}_i\right) \tag{A.14}$$

$$= 0 + (w_1)^1 X_{i,1} - 0 \tag{A.15}$$

$$= 1 \times w_1^{1-1} \mathbf{X}_{i,1} \tag{A.16}$$

$$= X_{i,1}, \tag{A.17}$$

and the final result is thus

$$\frac{\partial}{\partial w_1} g\left(f(\mathbf{w})^{(i)}\right) = \frac{\partial}{\partial w_1} g(\mathbf{w}) \frac{\partial}{\partial w_1} f(\mathbf{w})^{(i)} \tag{A.18}$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(w_0 + w_1 \mathbf{X}_{i,1} - \mathbf{t}_i\right) \mathbf{X}_{i,1} \tag{A.19}$$

APPENDIX B

Code Snippets

This appendix holds various pieces of code that were introduced in the report.

B.1 GPU Information Script

The script presented in listing B.1 is the entire script which connects to the GPU servers and fetches the relevant information for the GPUs. Section 3.7.1 motivated why this script was necessary.

Listing B.1: Script to output GPU information from servers.

```
1 echo '  ======================  GPU status  ========================== '
2 echo 'Theia'; ssh theia nvidia-smi | grep '%' | sed 's/|/ /g'
3 echo 'Rhea'; ssh rhea nvidia-smi | grep '%' | sed 's/|/ /g'
4 echo 'Oceanus'; ssh oceanus nvidia-smi | grep '%' | sed 's/|/ /g'
5 echo 'Phoebe'; ssh phoebe nvidia-smi | grep '%' | sed 's/|/ /g'
6 echo 'Mnemosyne'; ssh mnemosyne nvidia-smi | grep '%' | sed 's/|/ /g'
7 echo 'Tethys'; ssh tethys nvidia-smi | grep '%' | sed 's/|/ /g'
8 echo 'Themis'; ssh themis nvidia-smi | grep '%' | sed 's/|/ /g'
```

B.2 t-SNE script

Listing B.2 present the code used to create the t-SNE plots which show where the model's attention lies.

Listing B.2: Script to create t-SNE plots.

```
import matplotlib.pyplot as plt
import numpy as np
import glob, os

# getting all paths for attention and bleu
attention = glob.glob('attention/char2word_to_char/*')
bleu = glob.glob('bleu/char2word_to_char/*')
pair = zip(attention, bleu)
char = False
no = 100 # which sentence in the batch to take
for a, b in pair:
    # making paths
    source_path = os.path.join(b, 'source.txt')
    translated_path = os.path.join(b, 'translated.txt')
    attention_path = os.path.join(a, 'attention.npy')
    # loading data
    with open(source_path, 'r') as f:
        source = f.read().split('\n')[no]
    with open(translated_path, 'r') as f:
        translated = f.read().split('\n')[no]
    # setting up data
    column_labels = unicode(source, 'utf-8')
    column_labels = list(column_labels) if char else column_labels.split(' ')
    column_labels += ['<EOS>']
    row_labels = list(unicode(translated, 'utf-8')) + ['<EOS>']
    att = np.load(attention_path)[no].T
    att = att[:len(column_labels), :len(row_labels)]
    # plotting
    fig, ax = plt.subplots()
    heatmap = ax.pcolor(att, cmap=plt.cm.Blues)
    ax.xaxis.tick_top()
    ax.invert_yaxis()
    ax.set_xticks(np.arange(att.shape[1])+0.5, minor=False)
    ax.set_yticks(np.arange(att.shape[0])+0.5, minor=False)
    ax.set_xticklabels(row_labels, minor=False)
    ax.set_yticklabels(column_labels, minor=False)
    plt.show(heatmap)
```

APPENDIX C

Project Documentation

For this appendix we sought our to document our communication with our supervisors. We include the newsletter we sent to our supervisors throughout the project documenting our communication in appendix C.1. Moreover we present a few relevant charts documenting from the project in appendix C.2.

C.1 Monthly Summaries

This section holds our updates which we would send to our supervisor on a weekly basis. We present these here for each month (from February to July – starting with C.1.1, C.1.2, C.1.3, C.1.4, C.1.5, and C.1.6).

C.1.1 February

```
Update 01 - 19 feb. 2016

Rapport:
- Fået oprettet kapitler og fået et overblik over hvad
   rapporten skal indeholde:

Indholdsfortegnelse

Introduction:
 - Motivation
Literature Review
 - (Project plan)
Methods
 - Super-/Unsuper-/semisupervised learning
 - Linear Regression
 - Logistic regression
 - Backpropagation
 - Neural Networks
   - Perceptron
   - Multi-layer perceptron
   - Convolutional neural networks
   - Recurrent neural networks
```

```
     - Long-short term memory
     - Gated recurrent units
   - Autoencoders
     - Variational autoencoders
   - Embeddings
Tools
 - GPGPU
 - Theano/TensorFlow
Results
 - (Projekt point 1,2, ... , 6)
Conclusion

Indtil videre har vi kun skrevet lidt om linear regression.

Tensorflow coding:

- Many-to-one vanilla RNN works
- Gather operation to do a char-2-word works (pulls out spaces
   from the RNN sequence successfully)
- Tested Tensorboard successfully
- Contributed to Tensorflow (github.com/tensorflow/tensorflow/
   blob/master/RELEASE.md)
```

```
Update 02 19-26 feb 2016

Code
A, more than "just" a data loader, data loader under
   developement (hoping to finish up this weekend so we can
   start getting some results..!)
github.com/alrojo/Frostings

Report
Started working on several sections of fundamental theory in
   the report(linear-, logistic regression, backpropagation etc
   .)
```

C.1.2 March

```
Code:
- first working version of frostings (our general purpose
   dataloader)
- europarl specific text_loader (on the frostings library)
   which supports all of our needs (length of sentences etc.)
```

```
- dictionary encoding of europarl (330+ symbols, including end
   of sequence, using same dict for both .fr and .en)
- tensorboard implemented
- t-sne implemented
- saving checkpoints of weights implemented
- prototype for sequence-to-sequence model works (though
   encoder parses all zeros)
- translation from one-hot output to chars (so we can read the
   network's predictions)
- ported all code to python3

Report:
No significant progress
```

```
Code:
- docker image for GPU
- accuracy score
- fixed a bug where the target label was given at prediction
   point and was not delayed (found when testing the uppercase
   dummy dataset).
- found that many of TensorFlows popular optimizers (rmsprop,
   momentum) does not support sparse matrix multiplication (yet
   ) which we use in the embeddings. One the basis of this we
   had a discussion on whether or not to abandone TF for theano
   /lasagne. However, as we can understand sparse matrix
   multiplication is not even supported by theano, the fix
   seemed doable (by us) and other optimizers such as adam
   worked perfectly we chose to continue with tensorflow.
- switched our current model (100 rnn units, sgd, alpha=0.1) to
    (100 gru units, adam, alpha=0.001).
- 95% accuracy after 11000 iterations on Theia (GPU) (32 *
   11000 examples) in all dummy datasets. Acting as expected.
- overfitted a small part of the europarl dataset (~2000
   sentences) with upto 95% train accuracy on Theia (
   significant milestone!)
- dummy dataset (as shown in the presentation monday the 7'th
   march 2016)

Report:
No significant progress
```

```
Code:
- validation/test split generator
```

```
- improvements to data loader (to supports lists of datasets
   and validation/test splits)
- improvements to data loader (splits dataset into multiple
   splits based on sequence length, to have size-wise shuffling
   )
- requirements.txt for project
- configurable sequence length of dataset (max amount of chars
   allowed for a sentence)
- excluding very rare characters from dataset
- L2 regularization
- prediction clipping
- gradient clipping
- accuracy computation on the GPU (using TF)
- decoder to use prediction instead of labels (significant
   milestone!)
- TensorBoard to compute total variance over weight, bias,
   gradients
- Switched to Danish translation (easier to debug)

Report:
No significant progress
```

```
Code:
- train.py configurations (Stop after N iterations, batch_size,
name for config file)
- display time per iteration
- postprocessing of decoded output (cutting away <EOS> and
everything after)
- postprocessing of decoded output (nltk to tokenize string for
word-wise BLEU)
- Edit/Levinstein distance of chars
- BLEU on sentence level own implementation. (significant!)

Report:
No significant progress
```

C.1.3 April

```
Code:
- Improved configuration architecture (as mentioned in meeting)
- Truncate training set instead of removing (100,000 ->
   2,000,000 sentences on europarl)
```

```
- Configuration architecture ala. Sander Dieleman's kaggle-ndsb
    (with minor improvements)
- TensorFlows seq2seq model (word2word), 0.13-0.16 bleu on
   Europarl.
- TensorBoard summeries on validation
- Validation debugging architecture testing and working (
   significant!)

Report:
No significant progress
```

```
Code:
- Scrutinized large parts of dataloader
- Rewritten larger parts of dataloader
- Made a warm-up scheduler (e.g. train first 10k iterations on
   small sentences.)
- Training several models.

Models: Char2Word encoder w. attention.
All trained (still training) with 15 epochs

seq_len = 50
Trains at 5 epochs/day
At 8 BLEU after 12 epochs (still increasing)

seq_len = 75
not running yet (in queue after seq_len=50)

seq_len = 100
Trains at 2 epochs/day
At 2.8 BLEU after 4 epochs (still increasing)

seq_len = 150
Trains at 2 epochs/day
At 2.4 BLEU after 4 epochs (still increasing)

seq_len = 200
Trains at 1.5 epochs/day
At 2.2 BLEU after 3 epochs (still increasing)

seq_len = 300
Trains at 1 epoch/day
At 1.8 BLEU after 2 epochs (still increasing)
```

```
Social media:
- https://github.com/harvardnlp/seq2seq-attn/issues/16

Report:
- Literature review
```

```
Code:
- Implemented char swapping to introduce noise, such that swap
   ("<SOS>Moon is blue", swap_percentage=0.05) -> "<SOS>Mion os
    blue"
- Implemented schedule for changing percentage of char swapping
    during training.
- Implemented Char2Word encoder architecture

Ran models with char swapping on T_decoder, char2word on
   encoder, different embeddings for enc/dec and larger models.
    All failed.

Social media:
- https://www.reddit.com/r/MachineLearning/comments/4egcgt/
   question_neural_machine_translation_char2char/

Key reads:
- Ling et al. 2015 (char -> char, ~20 Bleu) http://arxiv.org/
   abs/1511.04586
- Chung et al. 2016 (BPE -> char, ~20-25 Bleu) http://arxiv.org
   /abs/1603.06147
- Sennrich et al., (BPE definition) http://arxiv.org/abs
   /1508.07909

Report:
No significant progress
```

C.1.4 May

```
Note for next week
- Test setups and hyperparameters
- Make a char-word-char and char-char baseline
- Finish methods section

CODE:
```

```
- Fixed bug with <SoS> missing when decoder was predicting,
   gave ½ better BLEU (6.5 at seq_len=50 on EN-DA europarl)
- Implemented mask
- Implemented bi-directional encoder (still under testing),
   gave 6.5 better BLEU (13.0 at seq_len=50 on EN-DA europarl)
- Minor improvements to data loader

TRAINING:
From now on we will provide traininglogs with the following
   speadsheet.

REPORT:
- Significant progress on Methods section
```

C.1.5 June

```
Note for next week
- Code cleaning
- Char-char baseline
- Get Moses BLEU and work on predictions/dataset (article
   standards)
- Finish methods section (is slightly delayed due to focus on
   model training)

-------------------
CODE:
- Splitted char embeddings (before encoder-decoder used the
   same embedding)
- Found our L2 was way too aggressive, setting it to 1e-10
   worked wonders (will consider completely removing it in
   future work)
- Tested linear+nonlinear combination on transfered state from
   encoder -> decoder (needs to be rerunned given "L2 bug")
- Tested A LOT of hyperparameters and setups
- Our code is approximately x2-x2.5 speed of tensorflows
   implementation. Mostly because we can handle dynamic
   sequences, but also because our loss implementation is
   vastly superior (handles 3D tensors as oppose to a list of 2
   D tensors)
- Fixed various bugs

TRAINING: (ran 32 models this week)
Traininglogs can be found in our speadsheet.
```

```
Of debugging, we are currently combatting model "explotion" and
    we will soon try to implement Casper's proposal of
   penalizing variance between hidden states.

REPORT:
- Significant progress on Methods section (finalizing stages)

IMPORTANT NOTES:
Our model on seq_len=500 on French europarl scored 20 BLEU. (
   http://imgur.com/sNXC47Y)
The best current char-char performance on French europarl is
   19.49 by Ling. et al. 2015 (http://arxiv.org/abs/1511.04586)
   .
However, as we are using a custom BLEU calculator we should
   implement the Moses BLEU before arriving at any conclusions.
```

```
Note for next week
- Writing, writing and more writing ..!
- Getting WMT'15 setup
- Test Moses BLEU on a seq_len=500

-------------------
CODE:
- Cleaned code and made a new default model using our own RNN/
   seq2seq implementations:

Regularization parameter: 0.000001
Embedding size: 256
Warmup: 10,000
seq_len: 50
separate embeddings
visualize_freq: 100,000

validation frequency
summary frequency
log freq
- Now using larger char embeddings per default (went from 16 ->
    256, which tested best on validation on 16, 32, 64, 128,
   256, 512)
- Hyperparameter testing of batch_size, found larger batch
   sizes to give significant increase in learning time and max
   BLEU (went from 21 to 24.5 BLEU by going from 64 -> 1024 in
   batch size), however, we are running into memory issues so
```

```
   we can only test this on seq_len=50.
- Tested char2char model (no usage of spaces ad above) got 19
   BLEU after 5 epochs with seq_len=200, was still learning!
   but trained extremely slow due to attention on 200 timesteps
- Now using official Moses BLEU (multi-bleu.perl, as
   recommended by Casper), their BLEU calculation is approx
   mean=2, stddev=0.25 lower than ours (so between 1.5 and 2.5
   lower, mostly 2). Our 1024 model had 24.9 our BLEU and 22.8
   Moses BLEU on a validation run. We will go over to using
   Moses BLEU and not investigate why it is slightly smaller,
   we assume it is something with tokenizing.
(here are some of the translation with 22.8 moses BLEU: http://
   pastebin.com/cMirbTFX)


TRAINING: (ran 18 models this week)
Traininglogs can be found in our speadsheet.

REPORT:
- Significant progress on all sections

NOTE:
Please note that all "default/standard" results are performed
   on the spaces model, such that

encoder
char_encoder = rnn(X_embeddings)
spaces = gather(char_encoder, places_of_spaces)
spaces_enc_forward = rnn(spaces)
spaces_enc_backwards = rnn(spaces, backwards=True)
spaces_enc_total = concat(spaces_enc_forward,
   spaces_enc_backwards)

decoder
s0 = spaces_enc_total[-1] # getting last state for initial
   state of decoder
decoder = decode_with_attention(spaces_enc_total, s0,
   t_embeddings) # only using t=truth embeddings when training,
    else just using predictions
```

```
Note for next week
- Writing, writing and more writing ..!
- Running WMT'15 models
```

```
- Also working on multi-gpu, but it is difficult

-------------------
CODE:
- Setup WMT'15 with all preprocessing, now training (much more
   difficult)
- Made seperate alphabets so it is similar to published
   material
- Made TSNE of chars (attached as .png)
- Made attention plots (attached as .png)
- Tested using less attention units (found going from 400->10
   had no impact, new default is 30).

TRAINING: (ran ~30 models this week, mostly debugging, we do
   not log debugging runs)
Traininglogs can be found in our speadsheet.

REPORT:
- Significant progress on all sections, currently at 72 pages
```

C.1.6 July

```
Note for next week
- Writing, writing and more writing ..!
- Running WMT'15 models
- Debugging WMT'15 models

-------------------
CODE:
- Setting up data downloader and alphabet to support all of our
    datasets (europarl da/fr/de and wmt'15 de)
- Made WMT'15 our default model (to work on it more intensively
   )

TRAINING: (ran ~15 models this week)
Traininglogs can be found in our speadsheet.
- We found that we can reduce attention to only ~30 units
   without loss of performance.
- We can reduce num_units to 300 without loss in performance.
- Dropping learning rate during training (using ADAM) gave a
   slight performance increase (~0.5 BLEU).
- We are currently stuck with WMT'15 (only getting ~3-4 BLEU)
   but we think we know why. It seems like it is not using the
```

```
   attention part, so we will try and plot the attention
   activations.


REPORT:
- Significant progress on all sections, currently at 79 pages.
```

```
Note for next week
- Writing, writing and more writing ..!
- Running final models
- We intend to finish the report by the 16'th of July

-------------------
CODE:
- Made the most beautiful data loader
in the history of neural machine translation,
giving significant performance increase
(known in our report as: variable bucket schedule with
   fuzziness)
- Made identical to Chung et al. setup (our benchmark article)
- Made configs for all final models
(will run c-to-c and c2w-to-c)

TRAINING: (ran ~20 models this week)
Traininglogs can be found in our speadsheet.
- Got from ~4 BLEU to currently ~14 BLEU (havnt converged yet)
on WMT'15 (Chung is at ~20).


REPORT:
- Significant progress on all sections, currently at 83 pages.
```

C.2 Charts

In this section we present some charts that we found interesting. Figure C.1 illustrates the number of models that we experimented with during the project. Our experiments peaked at week 18, where we had more than 40 experiments run.

Throughout the project we kept sending updates to our supervisor in which we would regularly inform of our current best BLEU score. These updates are the newsletters presented in appendix C.1. In fig. C.2 we show the BLEU score development. Notice at the drop in BLEU score at the end. This is due to the fact that we switched to a much larger data set and with another target language than the previous experiments. We

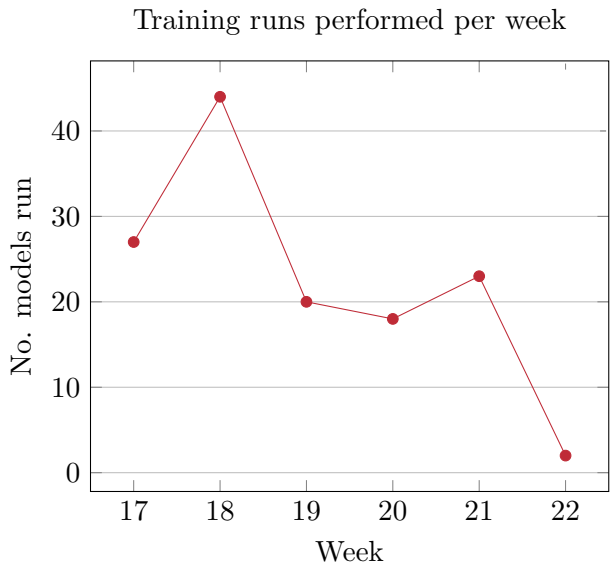

Figure C.1: Illustrate the number of experiments performed (the no. models run) from `https://docs.google.com/spreadsheets/d/1yftT-g7C5ah5Hhpl6P8AKeRZq8UxrDk65gsyaAmpmMk`.

started experimenting with the English to French data set and later switched to English to German.

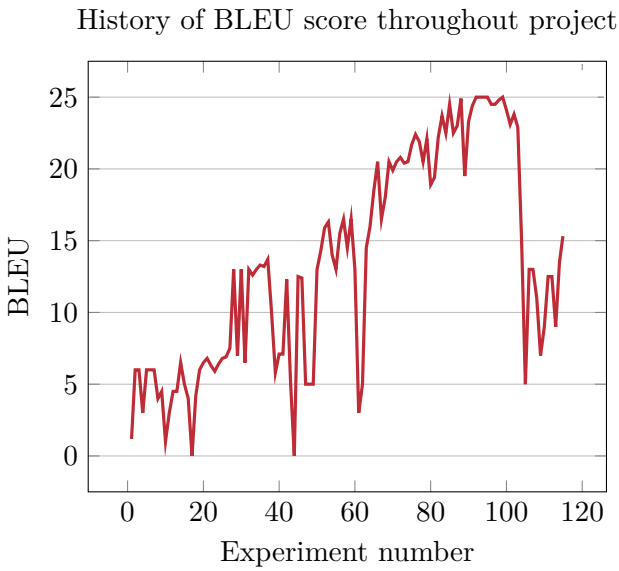

Figure C.2: Illustrate the development of BLEU score throughout the project.

Bibliography

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Jon Shlens, Benoit Steiner, Ilya Sutskever, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Oriol Vinyals, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015.

D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. Engel, L. Fan, C. Fougner, T. Han, A. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *ArXiv e-prints*, December 2015.

Stephen R. Anderson. How many languages are there in the world?, 2010. URL `http://www.linguisticsociety.org/content/how-many-languages-are-there-world`.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042, 2016. URL `http://arxiv.org/abs/1603.06042`.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *Iclr 2015*, 26(1): 1–15, mar 2014. ISSN 0147-006X. doi: 10.1146/annurev.neuro.26.041002.131047. URL `http://arxiv.org/abs/1409.0473http://www.annualreviews.org/doi/abs/10.1146/annurev.neuro.26.041002.131047`.

Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer, 2006.

Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 1st edition, 2006.

Ondrej Bojar, Christian Buck, Chris Callison-Burch, Barry Haddow, Philipp Koehn, Christof Monz, Matt Post, Herve Saint-Amand, Radu Soricut, and Lucia Specia, editors. *Proceedings of the Eighth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Sofia, Bulgaria, August 2013. URL `http://www.aclweb.org/anthology/W13-22`.

Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Christof Monz, Matt Post, and Lucia Specia, editors. *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Baltimore, Maryland, USA, June 2014. URL `http://www.aclweb.org/anthology/W/W14/W14-33`.

Ondrej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina, editors. *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Lisbon, Portugal, September 2015. URL `http://aclweb.org/anthology/W15-30`.

Randal E Bryant and O'Hallaron David Richard. *Computer systems: a programmer's perspective*. Prentice Hall Upper Saddle River, 2nd edition, 2010.

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv e-prints*, December 2015.

KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014a. URL `http://arxiv.org/abs/1409.1259`.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. jun 2014b. URL `http://arxiv.org/abs/1406.1078`.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014c. URL `http://arxiv.org/abs/1406.1078`.

Junyoung Chung, Kyunghyun Cho, and Yoshua Bengio. A character-level decoder without explicit segmentation for neural machine translation. *CoRR*, abs/1603.06147, 2016. URL `http://arxiv.org/abs/1603.06147`.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011a.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011b.

Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016. URL `http://arxiv.org/abs/1603.09025`.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, December 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL `http://www.deeplearningbook.org`.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016. URL `http://arxiv.org/abs/1606.03401`.

Geoffrey E Hinton. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

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

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. *MT Summit*, 2005. URL `http://homepages.inf.ed.ac.uk/pkoehn/publications/europarl-mtsummit05.pdf`.

David Krueger and Roland Memisevic. Regularizing rnns by stabilizing activations. *CoRR*, abs/1511.08400, 2015. URL `http://arxiv.org/abs/1511.08400`.

Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015. URL http://arxiv.org/abs/1506.07285.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation. aug 2015a. URL http://arxiv.org/abs/1508.02096.

Wang Ling, Isabel Trancoso, Chris Dyer, and Alan W. Black. Character-based Neural Machine Translation. nov 2015b. URL http://arxiv.org/abs/1511.04586.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective Approaches to Attention-based Neural Machine Translation. page 11, aug 2015a. URL http://arxiv.org/abs/1508.04025.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015b. URL http://arxiv.org/abs/1508.04025.

André F. T. Martins and Ramón Fernández Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. *CoRR*, abs/1602.02068, 2016. URL http://arxiv.org/abs/1602.02068.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013b.

Thomas M Mitchell. Machine learning. *Machine Learning*, 1997.

J Moody, S Hanson, Anders Krogh, and John A Hertz. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4:950–957, 1992.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002a. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL http://dx.doi.org/10.3115/1073083.1073135.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a Method for Automatic Evaluation of Machine Translation. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, pages 311–318, 2002b. ISSN 00134686. doi: 10.3115/1073083.1073135. URL http://portal.acm.org/citation.cfm?doid=1073083.1073135.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012. URL http://arxiv.org/abs/1211.5063.

Baolin Peng, Zhengdong Lu, Hang Li, and Kam-Fai Wong. Towards neural network-based reasoning. *CoRR*, abs/1508.05508, 2015. URL http://arxiv.org/abs/1508.05508.

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

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. ISSN 0028-0836. doi: 10.1038/nature16961. URL http://dx.doi.org/10.1038/nature16961.

Søren Kaae Sønderby and Ole Winther. Protein secondary structure prediction with long short term memory networks. *arXiv preprint arXiv:1412.7828*, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. page 9, sep 2014. URL http://arxiv.org/abs/1409.3215.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013. URL http://arxiv.org/abs/1312.6199.

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.

Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

Aäron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *CoRR*, abs/1601.06759, 2016. URL http://arxiv.org/abs/1601.06759.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37.

Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Trans. Evol. Comp*, 1(1):67–82, April 1997. ISSN 1089-778X. doi: 10.1109/4235.585893. URL `http://dx.doi.org/10.1109/4235.585893`.

David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. *CoRR*, abs/1602.08210, 2016. URL `http://arxiv.org/abs/1602.08210`.