Master's Programme in Industrial Engineering and Management

MatsuLM - neural network language modeling toolkit

Python implementation of a neural network language modeling toolkit

---

Riko Nyberg

MASTER'S
THESIS

Aalto University
MASTER'S THESIS 2020

MatsuLM - neural network language modeling toolkit

Python implementation of a
neural network language modeling toolkit

Riko Nyberg

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, 31 July 2020

Supervisor: professor Mikko Kurimo
Advisor: postdoctoral researcher Mittul Singh

Aalto University
School of Science
Master's Programme in Industrial Engineering and Management

Author
Riko Nyberg

Title
MatsuLM - Python implementation of a neural network language modeling toolkit

School School of Science

Master's programme Industrial Engineering and Management

Major Leadership and Knowledge Management **Code** SCI3048

Minor Analytics and Data Science **Code** SCI3073

Supervisor professor Mikko Kurimo

Advisor postdoctoral researcher Mittul Singh

Level Master's **Date** 31st of July 2020 **Pages** 48 + 6 **Language** English

Abstract

Language models (LMs) give a probability of how likely a sequence of words might appear in a particular order in a sentence, and they are an essential part of automatic speech recognition (ASR) and natural language processing (NLP) systems. These systems have improved at a considerable pace over the past decade. Similarly, language models have significantly advanced after the invention of recurrent neural network language models (RNNLMs) in 2010. These RNNLMs are generally called neural network language models (NNLMs) and they have become the state-of-the-art language models because of their superior performance compared to N-gram models.

This thesis is creating a new NNLM toolkit, called MatsuLM, that is using the latest machine learning frameworks and industry standards. Hence, it is faster and easier to use and set up than the existing NNLM tools. Currently, there are very few open-source toolkits for NNLMs; however, these toolkits have both become outdated and are no longer supported, or they suffer from functionality issues.

This work introduces a new NNLM toolkit, called MatsuLM, that includes all the essential components to create and monitor NNLM development effortlessly. This toolkit is built to be as lightweight and straightforward as possible to decrease development effort in the future.

MatsuLM's performance is compared against two existing NNLM toolkits (TheanoLM and awd-lstm-lm). In the experiments conducted during this thesis, both existing toolkits were slower than the newly presented MatsuLM in training the language models. Consequently, MatsuLM is currently the fastest and most up to date NNLM toolkit compared to TheanoLM and awd-lstm-lm.

Keywords neural networks, language modeling, machine learning, deep learning, pytorch, LSTM

url https://riko.io/matsulm.pdf

Tekijä
Riko Nyberg

Työn nimi
MatsuLM - Python implementation of a neural network language modeling toolkit

Korkeakoulu Perustieteiden korkeakoulu

Master's programme Industrial Engineering and Management

Major Leadership and Knowledge Management **Code** SCI3048

Minor Analytics and Data Science **Code** SCI3073

Valvoja professori Mikko Kurimo

Ohjaaja tutkijatohtori Mittul Singh

Työn laji Diplomityö **Päiväys** 31.07.2020 **Sivuja** 48 + 6 **Kieli** englanti

Tiivistelmä

Kielimallit (LM) antavat arvion siitä, kuinka todennäköisesti sanasarja saattaa esiintyä lauseessa. LM on olennainen osa automaattista puheentunnistus- (ASR) ja luonnollisen kielenkäsittelyn- (NLP) järjestelmiä. ASR ja NLP järjestelmät ovat parantuneet huomattavalla vauhdilla viimeisen vuosikymmenen aikana. Samoin kielimallit ovat kehittyneet huomattavasti sen jälkeen kun ensimmäiset rekursiivisien neuroverkkojen kielimallit (RNNLM) kehitettiin vuonna 2010. Näitä RNNLM-malleja kutsutaan yleensä neuroverkkokielimalleiksi (NNLM) ja niistä on tullut alan huipputeknologiaa erinomaisen suorituskykynsä vuoksi.

Tämä opinnäytetyö esittelee uuden NNLM-työkalun, nimeltään MatsuLM, joka käyttää uusimpia koneoppimisstandardeja sekä -komponentteja. MatsuLM on nopeampi ja helpompi käyttää ja asentaa kuin nykyiset NNLM-työkalut, sen modernin koneoppimisstandardisoinnin vuoksi. MatsuLM sisältää kaikki tärkeät komponentit vaivatonta NNLM-kehitykstä varten. MatsuLM työkalu on rakennettu avoimella lähdekoodilla olemaan mahdollisimman kevyt ja helppokäyttöinen, jotta sen avulla voidaan vähentää NNLM-kehityksen työläyttä tulevaisuudessa.

Motivaatio MatsuLM:n kehitykseen syntyi NNLM-kehitykseen tarkoitettujen avoimella lähdekoodilla tehtyjen työkalujen puutteesta. Lisäksi olemassaolevat avoimen lähdekoodin NNLM-työkalun kärsivät puutteellisesta toiminnallisuudesta, koska ne ovat vanhentuneita, eikä niitä enää tueta tai ne ovat vielä varhaisessa kehitysvaiheessa, eivätkä siksi ole luotettavia.

MatsuLM:n suorituskykyä verrataan kahteen muuhun olemassa olevaan NNLM-työkaluun (TheanoLM ja awd-lstm-lm). Opinnäytetyön aikana suoritetuissa kokeissa molemmat vertailussa käytetyt NNLM-työkalut olivat hitaampia, kuin MatsuLM NNLM-mallien koulutuksessa. Näin ollen MatsuLM on nopein ja pävitetyin NNLM-työkalu verrattuna TheanoLM:ään sekä awd-lstm-lm:iin.

Avainsanat neuroverkot, kielimallinnus, koneoppiminen, syväoppiminen, pytorch, LSTM

url https://riko.io/matsulm.pdf

Contents

Abbreviations

ASR	Automated Speech Recognition
NLP	Natural Language Processing
NN	Neural Network
FFNN	Feedforward Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-term Memory
LM	Language Model
SLM	Statistical Language Model
ML	Machine Learning
NNLM	Neural Network Language Model
PPL	Perplexity
GPU	Graphical Processing Unit
<unk>	unknown

1. Introduction

This thesis introduces a new toolkit for building neural network language models (NNLMs). Language models (LMs) give a probability of how likely a sequence of words might appear in a certain order in a sentence. LMs are applied to a wide range of modern natural language processing (NLP) applications. LMs are important for automatic speech recognition (ASR), machine translation, and spelling correction systems, to name a few. LMs form a significant part of NLP applications because many tasks depend on the quality of the LMs used. The quality of LMs is determined with perplexity (PPL), which tells how well they can predict the correct sequence of words. (Chelba et al. 2013)

These days, computational load is no longer as a significant limitation as it has been for research and development of language models, Automatic Speech Recognition (ASR), or Natural Language Processing systems. The technology has evolved rapidly by making computers, and especially Graphics Processing Units (GPUs), more powerful, cheaper, and hence, more accessible (You et al. 2019; Chen et al. 2014; Colic, Kalva, and Furht 2010). However, this computational load is still a limitation for the training of LMs. This limitation is one reason for building a new tool in this master's thesis, MatsuLM, to reduce the computational load of language model training with the latest machine learning frameworks. Reducing the computational load can be done by using GPUs more optimally so that the training times of LMs are reduced.

The computational processing power is essential for LMs and, in general, for most machine-learning applications because the quality of the LMs and other machine-learning models is affected by the quantity of the training data. In the case of LMs, the data quantity limitation often results from computational power. Training LMs with a large corpus becomes unworkable if it takes too much time to train only a single LM.

While developing LMs, there is often a need to perform test runs, and if these take days, then progress can become painfully slow. Other factors affecting the performance of LMs are the quality of the training data, the similarity of the testing and training data, and the way of estimating the performance and accuracy (e.g., perplexity) (Chelba et al. 2013).

Major companies already provide access to powerful computers without any monetary fee to use them (e.g., Google Colab). Nevertheless, despite these affordable or free resources, other factors hinder research in the field of NLP. These include adequate funding, people with necessary competence and knowledge, the availability of data for studies, and the tools necessary to perform analysis.

1.1 Motivation

The underlying motivation for this Master's thesis came from the Department of Signal Processing and Acoustics at Aalto University, Finland, who sought a functioning tool that matches their needs. The department has been using a toolkit called TheanoLM, which relies on the Theano Python library that is no longer supported or developed. Hence, updating the toolkit is evident for the Department of Signal Processing and Acoustics at Aalto University.

Over recent years, the author has worked with ASR and NLP systems, notably in the Apple AI team, developing a voice assistant called Siri, and has seen the rapid improvement in these systems during that time. In the course of his work, the author has developed personal preferences for using the most user-friendly tools and has listened to other users' opinions regarding their individual preferences. The Pytorch tool is considered to be the easiest for new Machine Learning (ML) developers to learn and use. For this reason, this thesis builds the newly presented toolkit on top of the Pytorch library.

Companies and researchers wish to have tools that facilitate efficient development. Naturally, these factors include cost efficiency, speed, and accuracy. The initial investment required to purchase the hardware necessary for performing machine learning, mostly GPUs that can train large language models, can be tens of thousands of euros. When the models and the training of these models are optimized, or when better tools are used, the initial investments can be significantly reduced to only a few hundred euros.

It is possible to train models for free by splitting up the model's training with proper tools to reduce the computational burden enough to run the training on free GPUs (e.g., Google Colab). This splitting up of the model's training gives the possibility to pause the training process, perform it on multiple devices simultaneously, or even move the training to another machine that can pick up where the previous machine left off.

These kinds of helpful language modeling tools are essential. For example, on the development side, shorter development and training times accelerate the whole development cycle noticeably. Moreover, faster language models shorten the response time on the production side, for example, for speech recognition products. Also, the current state-of-the-art tools all implement the latest algorithms. Hence, by using these state-of-the-art tools, one can ensure that the system is always up to date, and one can assume to have the best opportunity for training the most accurate machine learning models.

1.2 Objectives and research question

The research question of this thesis aims to create a modern open-source Python library – known as MatsuLM – that will make neural network language modeling (NNLM) faster and easier. This new NNLM toolkit overcomes the handicaps and limitations of existing toolkits. For instance, this tool provides researchers with a natural way of viewing training parameters and model architecture. The tool allows defining the neural network configuration in a separate file or as user inputs to a terminal command when starting the training. Modern methods of adding training parameters and model architecture provide a quick and easy way to update the neural network architecture or to set hyperparameters. The new toolkit incorporates user interface elements that have become standards in machine learning frameworks. When the toolkit is aligned with these standards, it will become more comfortable and faster for researchers and other users to learn to use and develop further.

1.3 Outline of the thesis

This Master's thesis consists of eight chapters. Chapter 1 introduces the research topic, giving the motivations and research questions of this thesis.

Chapter 2 provides some background to the subject of language modeling, and chapter 3 explains what neural network language models are. The research question is covered in chapters 4, 5, and 6: Chapter 4 describes the MatsuLM toolkit; chapter 5 explains the experimental setup of the new and existing NNLM toolkits; chapter 6 will go through the results of these experiments. Chapter 7 explains the contribution of this research to the collective community development of language modeling, and chapter 8 will point out some future work to be done related to the MatsuLM toolkit. An illustration of this thesis's outlines can be found in figure 1.1.

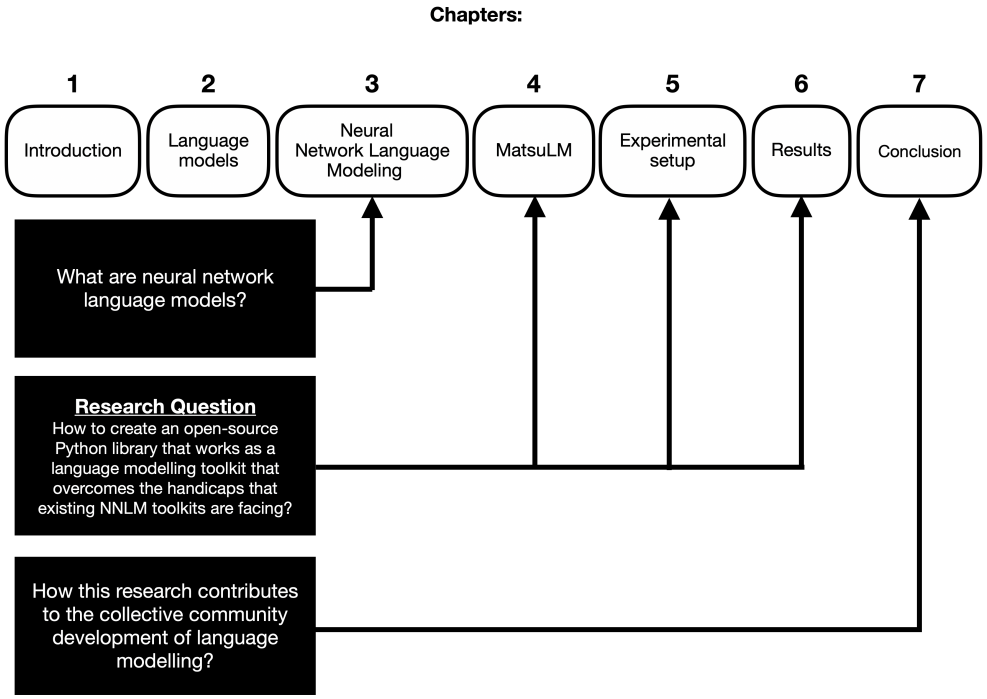


Figure 1.1. Outline of the Thesis

2. Language models

The field of natural language processing (NLP) has been developing rapidly over the past decade. Some of this development has resulted from increased computational capacity, such as larger memories and more powerful graphics processing units (GPUs). These have enabled larger datasets and ever more complex and computationally heavy calculations (Chen et al. 2014; Colic, Kalva, and Furht 2010; You et al. 2019). These computational performance improvements have also enabled researchers to use a much greater training corpus for language modeling (LM), a sub-field of NLP (Chelba et al. 2013).

One can think of language modeling as a task of giving a probability to given sentences. In practice, this would mean that when one inputs a sequence of words into a language model, the language model will output a probability of how likely these words will appear in the given order. This chapter will introduce the importance of language models in modern society and explain how the classical (non-neural) language models work.

2.1 Need for language models

Language modeling in modern times began in the 1980s when the first models of any significance were developed (Rosenfeld 2000). These first models were designed for written words, and, since then, the model has been adapted and improved to include the capture of spoken words. Today, language models are needed to accelerate and enhance the communication between humans as well as for the interaction between humans and computers. Some concrete and visible use cases for language models are intelligent keyboards, response suggestions for emails (Kannan et al. 2016), autocorrection for spelling, and virtual assistants.

2.1.1 Accelerating communication

The most visible language model implementation, with regards to the acceleration of communication between humans, is the autocompletion tool on a typical smartphone. Google already started using autocompletion in its search engine (figure 2.1) in 2004 to improve and accelerate the interaction between humans and computers.

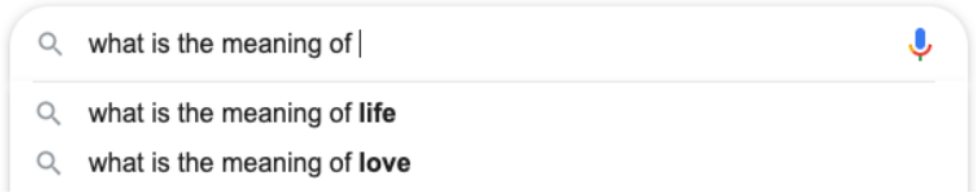
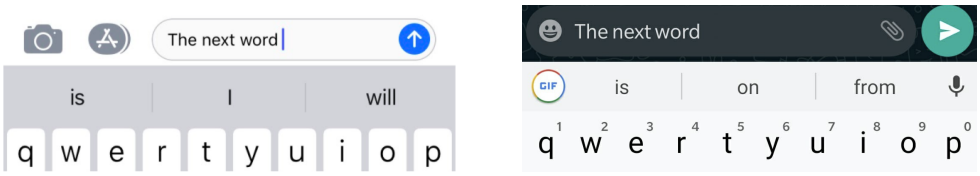


Figure 2.1. An example of accelerated human-computer interaction by Google

Nowadays, both Google and Apple are using language models in their software to predict subsequent words when writing messages (figure 2.2). This prediction of subsequent words can be done with so-called statistical language models or neural network language models (NNLMs). Today, however, the preference is more for the use of NNLMs due to their superior performance, which will be discussed in chapter 3.



(a) iPhone **(b)** Android phone

Figure 2.2. An example of accelerated interaction between people

2.1.2 Human-computer interaction

Language models play a critical role in human-computer interaction with automatic speech recognition (ASR) systems when ASR systems seek to understand the context of what the human is trying to say. For humans, this matching of speech or voice to correct words is easy because we have learned, through trial and error, to automatically match the correct context to words and phrases throughout our lives. However, computers lack the contextual knowledge that is necessary for effectively processing spoken communication. For example, if a human asks:

Can you hear me?

Now, as the pronunciation of “hear” and “here” are so similar, the ASR may inaccurately interpret the question as:

Can you here me?

This question would make no sense for a human being who possesses the background knowledge that helps to understand or "predict" the intended phrase. The language model is the tool that tells the speech recognition system, which of the given set of possible phrases, is the most probable. Hence, the LM is essential for the performance of a speech recognition system as it might not be sure which words have been said. For example, due to similar pronunciations, poor hearing of the voice, or loud background noise.

A language model can identify phrases that make no sense to a human speaker because they have learned the probability of sentences by seeing vast amounts of written text. A language model, trained with quality data, should not have seen a sentence "Can you here me?" but it has most probably seen some combination of a sentence "Can you hear me?". Hence, if a speech recognition system asks the language model which one of these is most likely to appear, it will give a higher probability to the "Can you hear me?" sentence. This way, LMs help ASR systems understand contextual information, which improves accuracy and performance in speech recognition. If speech recognition systems were not using a language model in deciding what has been spoken, it would generate much more sentences that would not make sense.

Some voice assistants can even show all possible interpretations of a human’s spoken phrase. For instance, Apple’s Siri is one such voice assistant, as shown in figure 2.3.

2.2 Classic language models

The history of language models is rich. It started from the classic approaches based on statistical language modeling, such as n-gram models that used different smoothing techniques to handle unseen n-grams (Kneser and Ney 1995). One recent summary of this history of language modeling is done by Popkes (2018). What follows here is based on her work and is supplemented with the work of Lankinen (2016). The au-

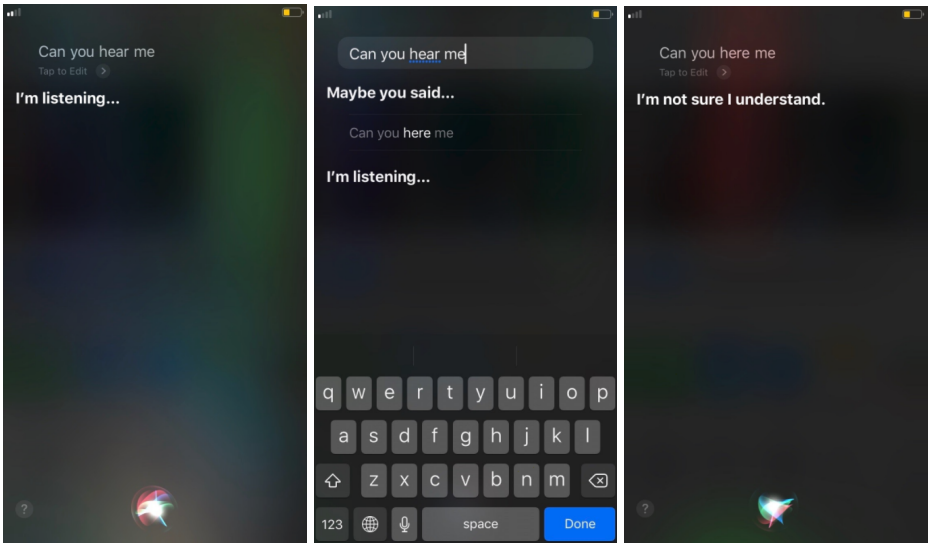


Figure 2.3. Possible interpretations of a spoken phrase

thor's present work builds upon these sources, adding to the information contained within them. However, some of the latest developments in the rapidly emerging field of language modeling are limited out of this work's scope. For example, this work does not discuss the transformer language models but suggest focusing on them in the future research (Radford et al. 2019; Devlin et al. 2018; Krause et al. 2019).

2.2.1 Statistical Language Modeling

One key function of NLP has been statistical language modeling, which is critical for speech recognition and machine translation (Rosenfeld 2000). Statistical language models aim to learn the probability $P(w_1, ..., w_n)$ of a sequence of words $w_1, ..., w_n$ (Tomáš Mikolov, Karafiát, et al. 2010; Goodman 2001; Jozefowicz et al. 2016). The chain rule of probability can be utilized to calculate this probability:

$$P(w_1, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_1, ..., w_{i-1}) \tag{2.1}$$

As there is much variation in the number of words that may precede a given word, and also due to the complexity of calculating $P(w_i | w_1, ..., w_{i-1})$ for many words i, the probability of a word is typically conditioned on a window of m previous words.

$$P(w_1, ..., w_n) \approx \prod_{i=1}^{n} P(w_i | w_{i-m}, ..., w_{i-1}) \tag{2.2}$$

There are numerous ways to use a language model. For example, the

model can anticipate and predict subsequent words; it can also issue probabilities to sentences. The following example demonstrates this. The language model may forecast that the sentence:

that is when I saw the three big giants walking towards me

has a greater likelihood of appearing in a text than the same sentence with a different ordering of the words:

walking big that saw is the me three when I giants towards

Among other things, this is used for tasks that recognize words in ambiguous contexts, like recognizing human speech, where the input is noisy. The following section 2.2.2 introduces one of the entrenched classical language models (called n-gram models) that have been used by the researchers for decades.

2.2.2 N-Gram Models

The N-gram model is a language model with low complexity, which is nothing more than a sequence of N words. For instance, a sequence of two words is a bigram (or 2-gram): *"I saw"* or *"walking towards"*. Adding a word to the sequence creates a trigram (or 3-gram) that contains three words; *"walking towards me"*. In a trigram model case, the probability of a sequence of words $w_1, ..., w_n$ would be calculated in the following way:

$$P(w_1, ..., w_n) \approx \prod_{n}^{i=1} P(w_i | w_{i-2}, w_{i-1}) \tag{2.3}$$

A trigram model can be generalized because it observes the two previous words in a given sequence, so it can be calculated as an N-gram that takes into account N-1 words.

$$P(w_1, ..., w_n) \approx \prod_{n}^{i=1} P(w_i | w_{i-N+1}, ..., w_{i-1}) \tag{2.4}$$

Here we use the Markov assumption, which is the term for the basic assumption that the probability of a word is only dependent on a limited number of previous words. An easy way of calculating trigram or N-gram probabilities is by using maximum likelihood estimation (MLE) (Jurafsky and Martin 2014). The estimate given by MLE for the N-gram probability

of a word w_i given a previous sequence of words $h = w_i|w_{i-N+1}, ..., w_{i-1}$ can be calculated by summing the number of times w_i appearances in the context h, and normalizing this by dividing every observations with h (Goodman 2001; Tomáš Mikolov 2012)

$$P(w_i|w_{i-N+1}, ..., w_{i-1}) = \frac{count(w_{i-N+1}, ..., w_{i-1}, w_i)}{count(w_{i-N+1}, ..., w_{i-1})} \tag{2.5}$$

For instance, consider that the words "*three big*" gives the context of those words h, and we wish to forecast the likelihood that the next word in the sequence w will be "*giants*". A training corpus offers a trigram model the ability to count the number of times "*three big*" was followed by "*giants*" and calculate:

$$P(w_i|w_{i-N+1}, ..., w_{i-1}) = \frac{count(\text{"}three\ big\ giants\text{"})}{count(\text{"}three\ big\text{"})} \tag{2.6}$$

Nonetheless, even N-gram models that have been trained with a large corpus are problematic. The reason for this is that it is challenging to calculate N-Gram probabilities. Like our previous example, numerous sequences of words typically appear very infrequently, only once, or not at all (Goodman 2001; Tomáš Mikolov 2012; Rosenfeld 2000). Let us consider the three-word sequence "*walking towards me*". What is the probability of having the word "*me*" following a sequence of words "*walking towards*"? A training corpus may contain not a single instance of that particular sequence. As a consequence, $count(walking\ towards\ me)$ would be zero, and hence, $P(me \mid walking\ towards)$ would also be zero. This is problematic as the sequence of words "walking towards" may appear a number of times in the corpus. Forecasting:

$$P(me \mid walking\ towards) = 0 \tag{2.7}$$

would fail to correctly estimate the actual likelihood of the sequence appearing.

Thus, the use of a standard N-gram model would yield such inaccurate zero probabilities on far too many occasions, making the model's predictions very noisy. Therefore, in order to circumvent probabilities of zero, it will be necessary to apply smoothing techniques. These smoothing techniques remove some probability mass from frequent events, redistributing it to unseen events. For example, those that have been assigned zero probability by the N-Gram model. (Goodman 2001; Jurafsky and Martin 2014; Tomáš Mikolov 2012)

There are significant issues related to the use of N-gram models, even though they are effective in certain contexts with a limited range of words and phrases. Modern recurrent neural network language models (RNNLMs) can offer improved perplexities and error rates in speech recognition systems compared to these traditional n-gram approaches (Tomáš Mikolov, Karafiát, et al. 2010; Tomáš Mikolov, Kombrink, et al. 2011; Adel, Vu, et al. 2013; Adel, Kirchhoff, et al. 2014). The following section introduces such RNNLMs.

3. Neural Network Language Modeling

The first neural network language model (NNLM) was introduced in 2001 by Bengio, Ducharme, et al. in the proceedings of Neural Information Processing Systems. Since then, researchers have investigated how to model language using neural networks. This research has accelerated over recent years due to many benefits that NNLMs possess, and due to increased computational performance that has made it possible to create bigger and more complex NNLMs (You et al. 2019; Chen et al. 2014).

The benefits that NNLMs possess are their ability to learn continuous word representations and complex relations by combining simple units in a hierarchy of non-linear layers. Other benefits include the fast performance in production and the model's generalization so that extremely small or zero probabilities would not be assigned to valid word sequences. (Arisoy et al. 2012).

These continuous word representations can be created with word embeddings, which are vector representations of the words. When properly creating the word embedding, semantically, or grammatically related words should be mapped to similar locations in the vector space. Therefore, the NNLMs can achieve good generalization of the model by giving realistic probabilities even for the unseen sentences or word combinations. (Pennington, Socher, and Manning 2014; Rong 2014; Arisoy et al. 2012). The downsides of using NNLMs and the word embeddings is that the training and creation require much computational power, it is slow, and it requires vast amounts of data. (Keselj 2019)

This first proposed large-scale NNLM by Bengio, Ducharme, et al. (2003) was based upon a simple feedforward neural network. Nevertheless, it was not until 2010 when Tomáš Mikolov, Karafiát, et al. introduced recurrent neural network language models (RNNLMs) that neural networks became established as a state-of-the-art technique for language modeling, replacing

the classic N-gram techniques. Neural networks have been proven to be exceptionally well-performing in this research field; for example, so-called transformer language models (Radford et al. 2019; Devlin et al. 2018; Krause et al. 2019).

The following chapter introduces the underlying architecture and functioning of artificial neurons, feedforward neural networks (FNNs), recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and how to evaluate the quality of these language models. The chapter also introduces different types of NNLMs: word-based, sub-word-based, and character-based.

3.1 Artificial and Biological Neurons

Artificial neurons are the building blocks of artificial neural networks. The idea behind artificial neurons is that they mimic biological neurons to a certain level, for example, the dendrites, cell bodies, or a nucleus and axons, as seen in the figure 3.1.

These biological functionalities are replaced in artificial neurons with mathematical models. However, artificial neurons are nowhere near as sophisticated as biological neurons, partly because the knowledge of the biological neurons is still limited. Hence, artificial neurons are simplified versions of their biological counterparts, for instance, by ignoring signal timing or the destruction and creation of new connections between neurons. Nevertheless, even these limited artificial models are sophisticated enough to solve simplified machine learning tasks.

An artificial neuron's simple abstraction of the biological neuron is the following. A neuron receives input from another neuron's output. An exception to this is the first neuron, called the input neuron, that takes in the input values for the network. Each connected neuron has a weight (w_n) that affects the previous neuron's output value before it reaches the neuron's input. This weight (w_n) can be positive or negative, and that determines how significant influence the first neuron has on the second one. With the weights (w_n), artificial neurons have a bias that acts as a threshold for the neuron's activation, either decreasing the threshold with a negative bias or increasing it with a positive bias.

The "input summation" (s) calculates all of these inputs (x_n) that are multiplied with their weights (w_n), and their products are summed and added to the bias. Mathematically it can be represented by the following

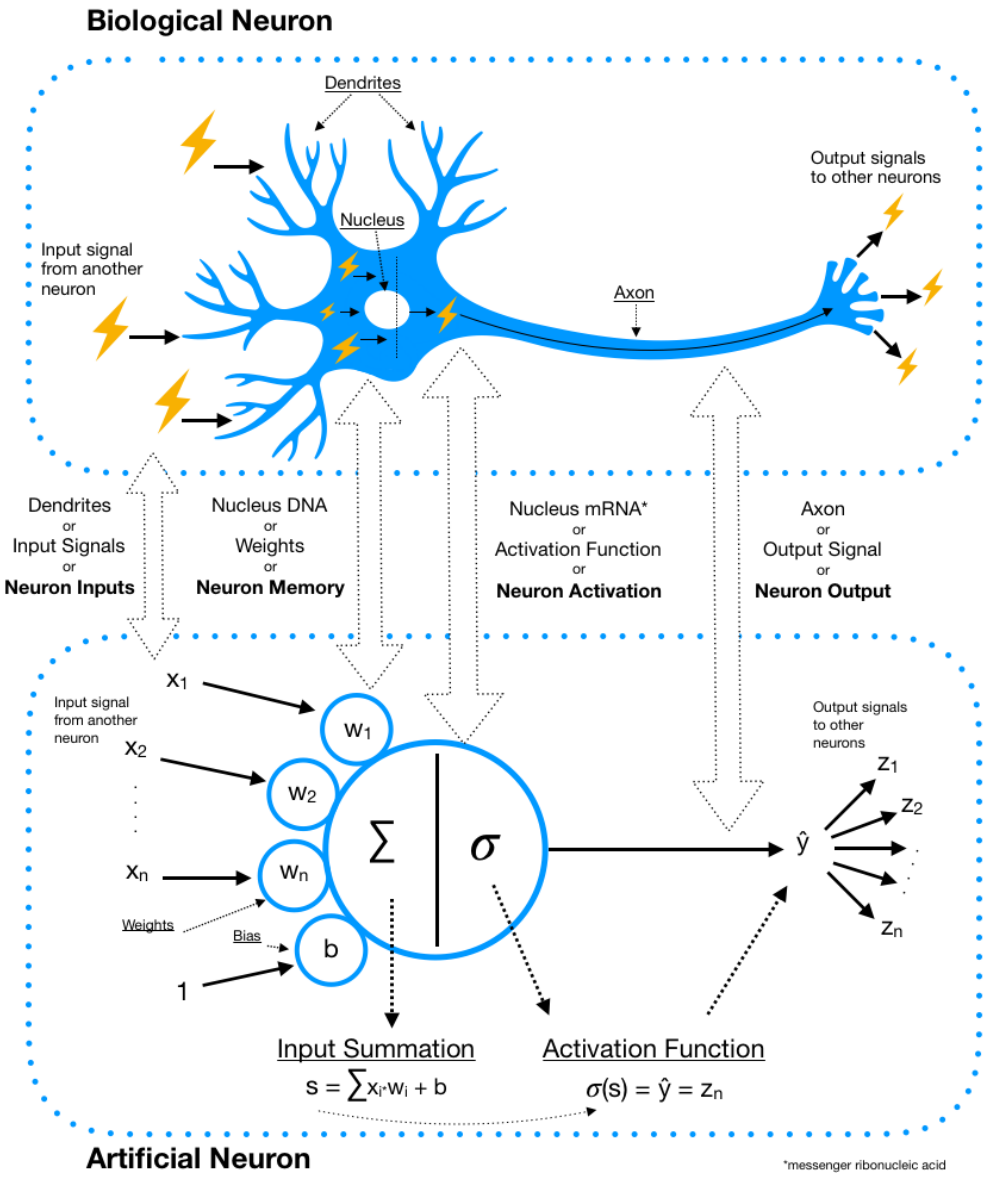


Figure 3.1. Comparison of biological and artificial neurons

equation:

$$s = \sum_{i=1}^{n} x_i * w_i + b \tag{3.1}$$

To make the notation simpler, we can add the bias (b) into the sum function. This can be done with an additional constant $x_0 = 1$ to the input vector and making the bias one of the weights ($w_0 = b$). Thus we can reformulate the previous equation 3.1 as:

$$s = \sum_{i=1}^{n} x_i * w_i = w^\top x \tag{3.2}$$

The neuron output is computed by adding this "input summation" (s) to

the neurons activation function σ.

$$\sigma(s) = \widehat{y} = z_n \tag{3.3}$$

The activation functions then decide what kind of signal will be sent forward to the following neurons. Hence, activation functions are also known as squashing functions, since they control the neuron's output limits. Some currently used activation functions are sigmoid, tanh, and ReLu, which limit the output to some specific range, like between -1 and 1, or to be greater or equal to zero. An illustration of a neuron with the mentioned properties can be found from figure 3.1.

3.2 Feedforward neural networks

A widely used definition for feedforward neural networks (FFNNs) is as follows, "A feedforward network defines a mapping $y = f(x; \theta)$ and learns the value of the parameters θ that result in the best function approximation" (Goodfellow, Bengio, and Courville 2016, p. 167). As the name of the model suggests, the information flows only in a single direction from input x to output y, and hence they are called feedforward networks. More precisely, in feedforward networks, there are no recurrent connections to feed the model's outputs back into itself. In its simplest form, the architecture comprises an input layer, at least one intermediate (hidden) layer and a prediction layer called an output layer. A simple illustration of a feedforward neural network (FFNN) with the mentioned properties can be found from figure 3.2.

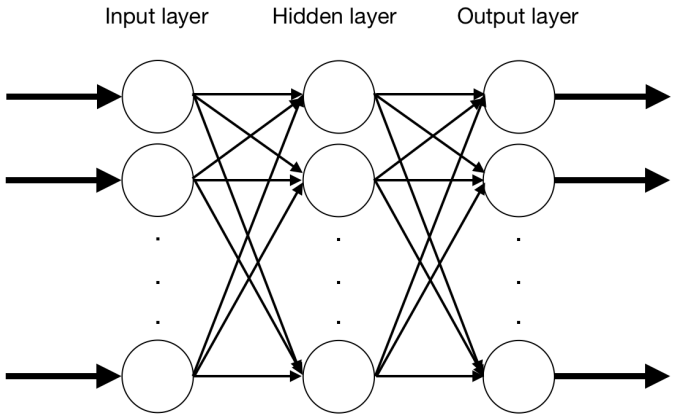


Figure 3.2. An example of feedforward neural network

Each layer is parameterized with some weights W and each of the layers activation function with a squashing function f. When the input values

would be x, this would mean that this network's output mapping would be the following:

$$f(x) = f^{output}\left(f^{hidden}(f^{input}(x))\right) \tag{3.4}$$

We can rewrite this function when we know that the network is fully connected, meaning that each layer's neurons are connected to all the neighboring layer's neurons. Rewriting can be done by using the output function of an artificial neuron from the previous section 3.1. When an artificial neuron's output is $\sigma(w^\top x)$, we can apply this for the whole layer by representing all the layer's weights as a weight matrix W. This way a whole layer of neuron outputs can be represented as a vector $\sigma(\mathbf{W}^\top x)$, making it possible to rewrite the mapping represented by the whole network as (Goodfellow, Bengio, and Courville 2016):

$$\begin{aligned} f(x) &= f^{output}(f^{hidden}(f^{input}(x)) \\ &= \sigma^{output}(W^\top_{output}(\underbrace{\sigma^{hidden}(W^\top_{hidden}(\underbrace{\sigma^{input}(W^\top_{input} * x)}_{\text{input layer output}}))}_{\text{hidden layer output}})) \end{aligned} \tag{3.5}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{output layer output}}$$

A drawback of FFNNs in language modeling is their inability to handle varying sequence lengths and the inability to retain the memory of earlier data. This drawback means that if an FFNN model took one word as an input, it would be almost impossible for the FFNN model to predict the word x_{t+1} in a sentence based on the one previous x_t word because the network can not know or remember the word x_{t-1} that came before the input word x_t. An example sentence could be "*How are you*". If trying to predict the last word of the sentence and the model is given two previous words, "*How are*", it is easy to say in this case that the possibility of next having the word "*you*" is a good one. On the contrary, if seeing only one preceding word "*are*", it is nearly impossible to predict what would follow because the options are vast. This example is illustrated in the figure 3.3, when x_t is the input at the time t and h_t is the output of the feedforward neural network (A_{ffnn}). The output is also called the prediction.

However, an FFNN can also be built to take more than one word as an input. In this case, the FFNN would have a local context constructed from the current input words. For example, if the FFNN input size would be two, it would be much more capable of predicting the "*you*" when the

input is "*How are*". This FFNN's ability to have local contexts is useful, for example, when creating word embedding, which is a way of mapping the relations between different words. However, even when having more that one word as an input, FFNNs are still unable to handle varying sequence lengths and retain the memory of earlier data, which is impractical for language models.

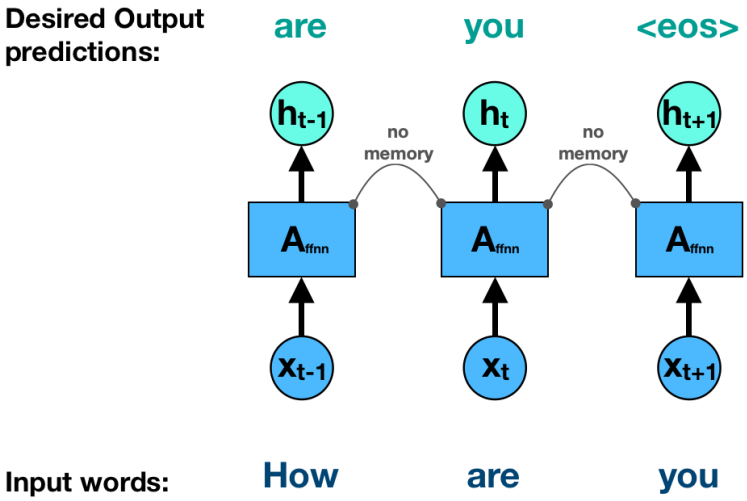


Figure 3.3. An example of feedforward neural network's drawback of handling inputs case-by-case (<eos> stands for end-of-sentence)

More detailed information about FFNNs can be found from the Deep Learning book by Goodfellow, Bengio, and Courville (2016).

3.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are intended for processing sequential data. Unlike feedforward neural networks, these network variants contain cyclic connections that can withhold the memory of previous inputs, as illustrated in figure 3.4. RNNs exist in multiple architectures, and their extreme versatility has been highlighted by Goodfellow, Bengio, and Courville (2016), "almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network."

Feedforward and recurrent networks differ mostly in how parameters are shared between different parts of a model. The sharing of parameters makes it possible to extend a model to examples of different lengths and generalize across examples. As Goodfellow, Bengio, and Courville (2016) indicate, the sharing of parameters is of great use in cases when the same information can be found at several positions within an input sequence.

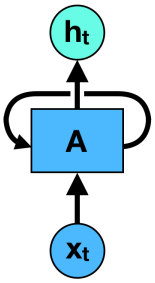


Figure 3.4. An example of a recurrent neural network's cyclic connections

To make their point, we can look at two sentences, "I climbed Mont Blanc in 2019" and "In 2019, I climbed Mont Blanc". When a machine learning model is being trained to extract the year of the activity, the model should be able to identify the year 2019 regardless of it appearing at the sixth or second position in the sequence of words comprising the sentence. This kind of learning would be complicated for traditional feedforward networks that process fixed-length sentences and contain different parameters for each input feature. Consequently, the FFNN model must learn, one by one, all the rules of the language in each sentence position. RNNs, therefore, offer greater time efficiency and performance by sharing the same weights throughout multiple time steps. (Goodfellow, Bengio, and Courville 2016)

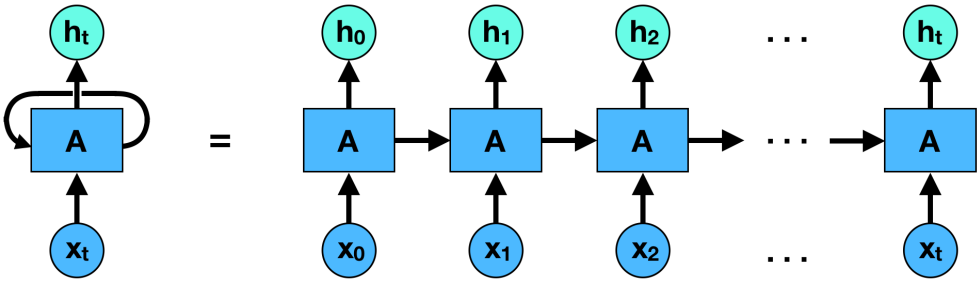


Figure 3.5. An example of unfolded recurrent neural networks cyclic connection

Parameter sharing is possible due to the operation of the RNN. When the output of a hidden unit is calculated, each component of the output is generated by applying the same update rule to each component of the previous output. This update can be illustrated by unfolding the RNN cyclic connection as in figure 3.5. The values of the hidden units at time step h_t, for an RNN with low complexity, can be described as follows (Goodfellow, Bengio, and Courville 2016):

$$h_t = f(h_{t-1}, x_t; \theta) \tag{3.6}$$

where h_{t-1} is the hidden state of the previous RNN time step, x_t is the input for the current time step, and θ is the value(s) used to parametrize

f for all used time steps. Here we can see that the hidden state h_t contains information about all the inputs because each hidden state uses the previous time step's hidden state as an input. This way, the RNN can include the whole input sequence past to every calculated hidden state, which created a sort of a "*memory*". This hidden unit that contains the information throughout multiple time steps is called a "*memory cell*" (Géron 2019; Goodfellow, Bengio, and Courville 2016). This memory cell makes RNNs great neural network models when the prediction can be improved by remembering the previous time steps. For instance, in the case of language models which might be used for predicting the subsequent word in a sentence.

3.3.1 Limitations

Simple Recurrent Neural Networks (RNNs) also have some significant drawbacks that appear, primarily when they are used in long sequences. When simple RNNs try to learn long-term dependencies, their performance is inadequate. The reasons for this are the gradients that tend to vanish or explode when propagated through multiple stages (Graves, Mohamed, and Hinton 2013; Bengio, Frasconi, and Simard 1993; Olah 2015).

To make this clearer, we can examine some examples. If we are trying to predict the last word in a sentence like "As the sun sets and darkness *falls*" or "Merry *Christmas*", we do not need more context to make good predictions because it is so obvious what the last words are most likely going to be. In these cases, the relevant information to predict the correct word is relatively close. Hence, a simple RNN works well in these cases, as illustrated in figure 3.6.

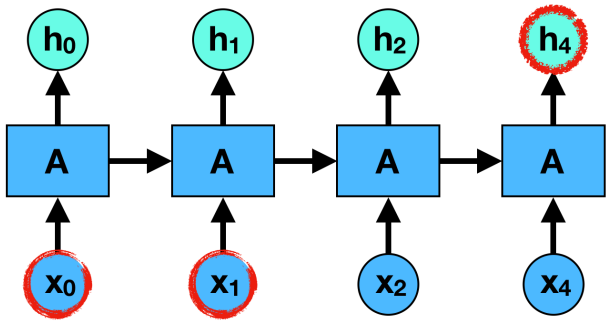


Figure 3.6. An example of a short Recurrent Neural Network connection

However, when the relevant words are further away from the predicted word, a simple RNN does not work as well. Let us consider these two sentences in a longer text "I've lived my whole life in Spain... I can fluently

speak *Spanish*." If we want to predict the last word here, we can predict from the last sentence that the last word will be a name of a language, but to know what the language is, we need to remember the previous sentences. Unfortunately, when it comes to these long-term dependencies, simple RNNs cannot learn them effectively (figure 3.7). (Bengio, Frasconi, and Simard 1993; Bengio, Simard, and Frasconi 1994)

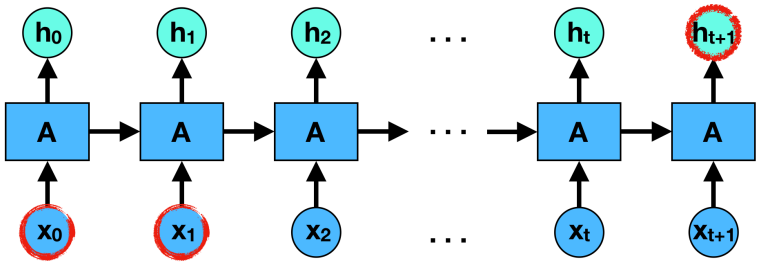


Figure 3.7. An example of a long Recurrent Neural Network connection

There are many ways to solve these long dependency problems. For instance, "skip connections" connects present and distant past variables (Lin et al. 1996). However, this thesis will concentrate on one special RNN architecture, called Long Short-Term Memory (LSTM), that is designed to solve the vanishing and exploding gradient descent problem. A more detailed description of RNNs can be found from the Deep Learning book by Goodfellow, Bengio, and Courville (2016).

3.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997) is among the most common neural network architectures, with convolutional neural networks (CNNs) (Fukushima 1980; LeCun et al. 1999) and the modern transformer models (Vaswani et al. 2017; Devlin et al. 2018), being used currently. LSTM is one of the most commonly used RNN architectures when it comes to machine translation (Sutskever, Vinyals, and Le 2014) or speech (Graves and Jaitly 2014; Graves, Mohamed, and Hinton 2013) and handwriting recognition (Carbune et al. 2020; Graves 2013; Graves and Schmidhuber 2009). LSTM is also used, for example, to help with motion (Ullah et al. 2017) and emotion recognition (Liu et al. 2018; Fan et al. 2016).

Long Short-Term Memory (LSTM) networks are built to overcome some of the problems faced by RNNs when trying to learn long-term dependencies. LSTM architecture design is more capable of finding and learning long-term dependencies and storing this information than standard RNNs

(Goodfellow, Bengio, and Courville 2016; Sak, Senior, and Beaufays 2014; Graves 2013). The LSTM network has been shown to give outstanding results in the previously mentioned tasks like machine translation, speech recognition, and handwriting recognition.

LSTM was introduced by Hochreiter and Schmidhuber (1997), and their work has been complemented and popularized by plenty of people. It is also essential to keep in mind that there are variations even between LSTMs, so the description that this thesis work will give is only one form of LSTM that is commonly used.

To better understand LSTMs, it is good to first look at some basic RNNs. Recurrent neural networks always have some repeating chain of neural networks. In the case of basic RNNs, this is something straightforward, like a single tanh neural network layer, as illustrated in figure 3.8.

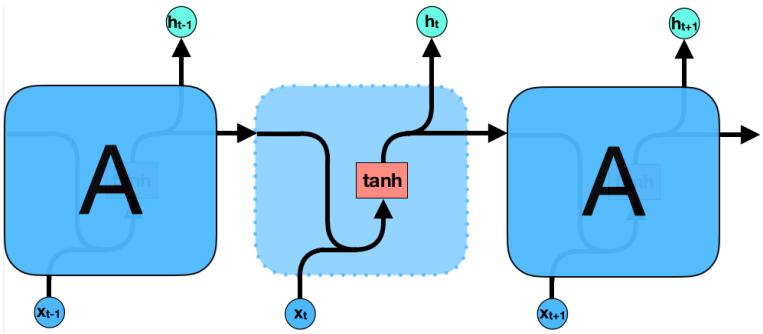


Figure 3.8. An example of a standard repeating Recurrent Neural Network module

When it comes to LSTMs, they also have this similar chain structure, but they are just more complicated by having four neural networks layers that interact with each other in a particular way (figure 3.9)

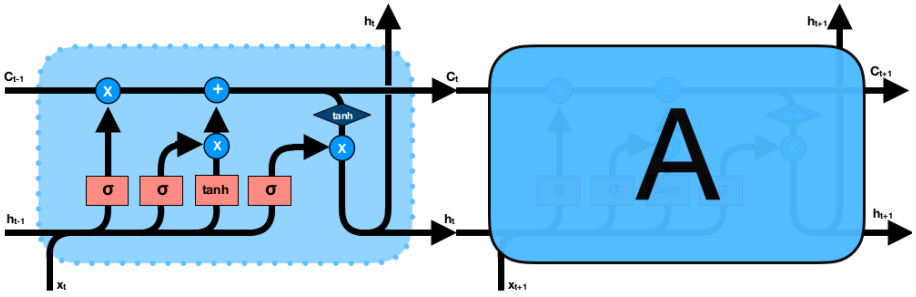


Figure 3.9. An example of LSTM modules (Olah 2015)

3.4.1 LSTM structure

This thesis deals with one of the commonly used types of LSTM structure and its implementation in Python. This section will introduce the structure of the particular LSTM implementation used in this thesis and explain its

functionality.

The core functionality of an LSTM network is called the "cell state" that goes through the LSTM network. Each LSTM module interacts with that cell state in three ways. First of all, it can increase or decrease the importance of some features that the cell state is carrying or storing in its memory. Secondly, it can add new values to the cell state. The third interaction is the copying of the cell state values. Those cell state values will be used in the making of the next prediction. In the case of language models, this could mean the predictions of the most likely following words. The structure of a single LSTM cell is illustrated in figure 3.10, and its cell state flow is highlighted with the three interaction stages that it has with the rest of the LSTM cell.

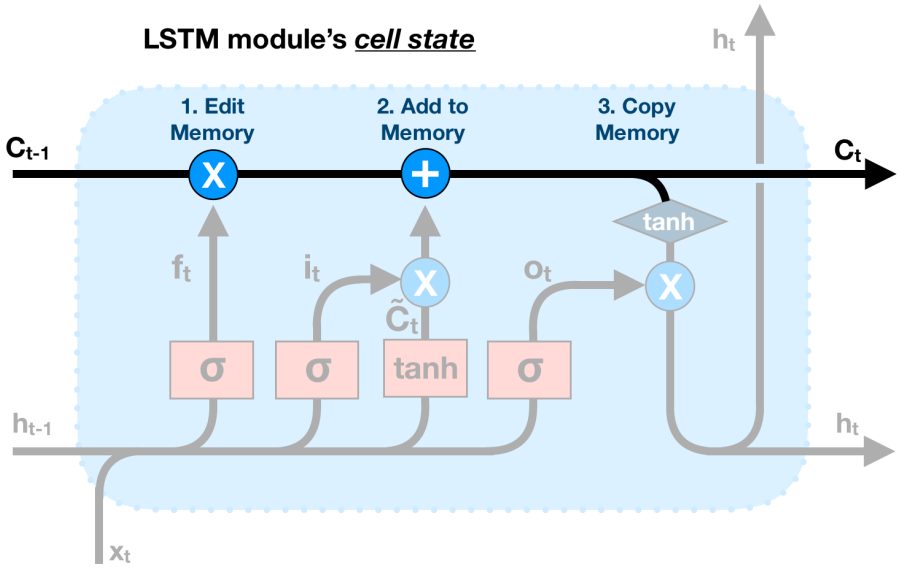


Figure 3.10. LSTM module's cell state

For example, if the last word or LSTM module input is a female name such as: "Then I saw Anna...", this might mean that this LSTM module that got the "Anna" as its input might want to modify the cell state to remember that there has been a mention of a female person. The LSTM network can then remember that it should refer instead to she and not he if there will be a reference to a person in the future. As an example, if the sentence would continue: "Then I saw Anna in the car waving _____ hand", the model can use the cell state to decide if the word should rather be her than his. Moreover, the LSTM module can also make the cell state forget things in the "Edit Memory" interaction. For example, if the previous text would continue as "... her hand to my brother, who had stopped because _____ was...". The missing word could now be either she or he, so the cell state should forget that it should be a she.

The first step of an LSTM module is illustrated in figure 3.11. There it gets the latest input (x_t), which is the latest word of the sentence, and the previous LSTM module's prediction (h_{t-1}), which is an encoded version of the word that the previous LSTM module's prediction of the next word. These words are represented in a vector format, which is a numeric representation of words so that they can be represented to a computer. These "word vectors" can also be trained so that similar words are close to each other in the vector space, making it possible for the computer to understand the similarity and relations of the words (Pennington, Socher, and Manning 2014; Rong 2014; Tomas Mikolov et al. 2013). This way, the words can be represented as a list of numbers, which is a vector, and these two vectors can be combined into a single long vector. In matrix terms, this is called concatenating, which is the process of joining one or more matrices to make a new matrix. In this case, we concatenate two one-dimensional matrices into a new bigger one-dimensional matrix. This concatenated input vector's notation is presented in the equation 3.7

$$V_{input} = \underbrace{[h_{t-1}, x_t]}_{\text{concatenation}} \tag{3.7}$$

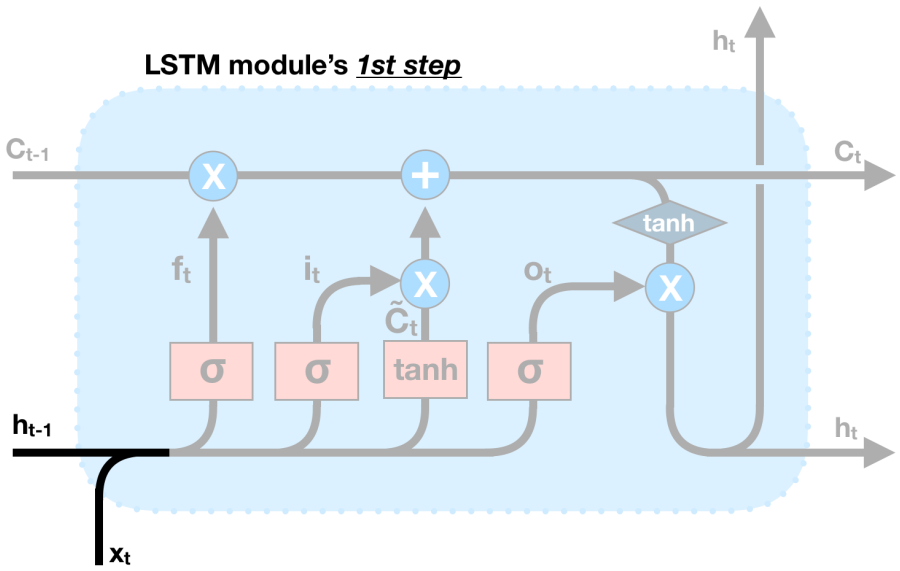


Figure 3.11. Concatenating the previous hidden state and the input

The second step, illustrated in figure 3.12, is where the LSTM module decides what will be forgotten from the cell state given the latest input and previous prediction information. So the previously concatenated vector (figure 3.11) is given as an input to the so-called "forget gate layer". The forget gate layer has a fully connected neural network layer that takes in the concatenated vector and outputs a vector size of the cell state memory.

The output of this fully connected neural network layer (f_t) is a list of numbers between 0 and 1, which is given to the so-called "forget gate". The forget gate does what it says, makes sure that the cell state memory is edited so that it forgets irrelevant info as illustrated in figure 3.10 and explained in the "she or he" case. The forget gate layer's (f_t) output values are multiplied in the forget gate (or "edit memory") parts with the cell state's corresponding values to create the "forgetting" affect in the cell state. When the forget gate value is closer to 0, it means that the cell state's corresponding value will be "more forgotten" if not totally and when the forget gate value is close to 1, the cell state's corresponding value will be "remembered". The mathematical notation of this forget gate's functions is presented in equation 3.8.

$$f_t = \sigma(W_f \cdot V_{input} + b_f) \tag{3.8}$$

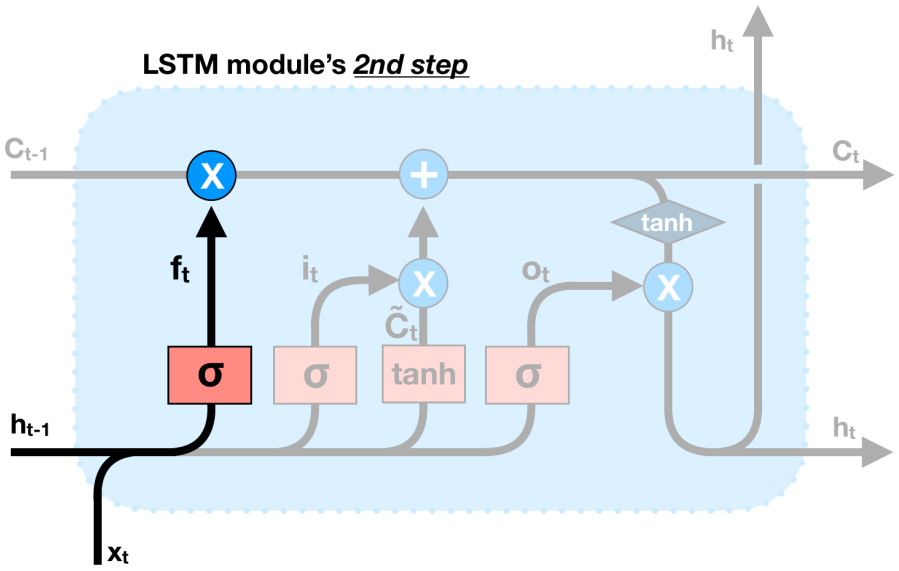


Figure 3.12. Forget gate

In the third step, which is illustrated in figure 3.13, the LSTM module decides what will be added to the cell state or, in other words, to the LSTM network's memory. Here the same concatenated list will be given separately as an input to two different networks where the first one creates a vector, called the "input gate" (i_t), that decides what new data is essential or irrelevant and hence, should be kept or forgotten. The second network, the so-called $tanh$ layer, uses the input vector to form new candidate values ($\widetilde{C}_t$) similar to the cell state. The candidate vector includes the candidates that would be added to the cell state, given the latest input (x_t) and previous prediction information (h_{t-1}).

However, before this new candidate vector ($\widetilde{C}_t$) values will be added to the cell state, the candidate vector ($\widetilde{C}_t$) will also be multiplied with a second forget gate, called the "input gate layer" (i_t). The input gate layer makes sure that only the relevant information is added to the cell state and that irrelevant information will be forgotten before it reaches the LSTM network's memory (cell state). The equation of calculating the input gate value i_t is illustrated in equation 3.9 and the candidate vector's ($\widetilde{C}_t$) calculation before multiplying it with the input gate values in equation 3.10.

An example of the added info to the cell state could be information about a person's gender, which was revealed in the last input.

$$i_t = \sigma(W_i \cdot V_{input} + b_i) \tag{3.9}$$

$$\widetilde{C}_t = tanh(W_C \cdot V_{input} + b_C) \tag{3.10}$$

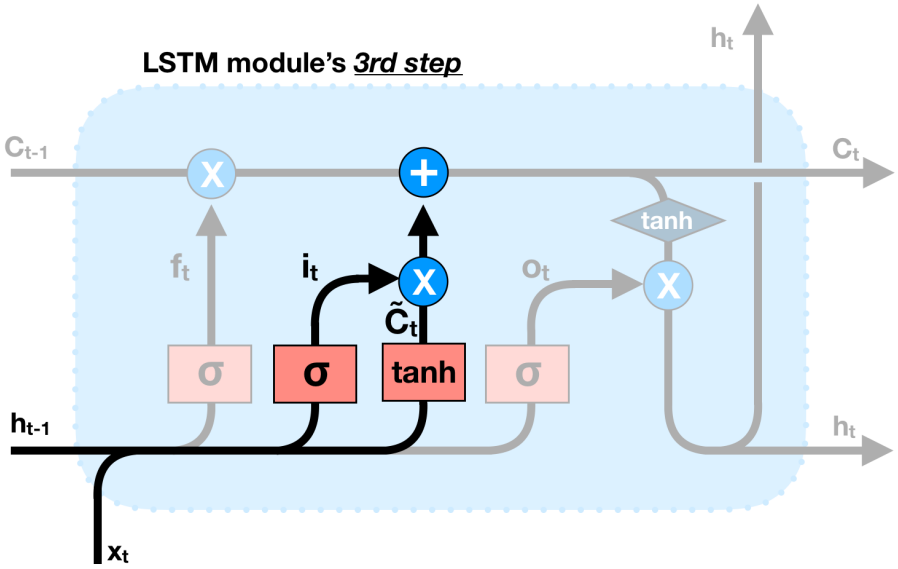


Figure 3.13. Input gate

The 4th step, illustrated in figure 3.14, is where the complete update of the LSTM network's memory is done. This update means calculating the new cell state C_t. To get the updated cell state, we will first multiply the previous cell state C_{t-1} with the forget gate f_t to see what we need to keep and remove from the LSTM network's memory. After this, we will add the candidate vector $\widetilde{C}_t$ that has also been multiplied with another forget gate called the "input gate" (i_t), which makes sure that only relevant

information is added to the cell state, the LSTM network's memory.

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t \tag{3.11}$$

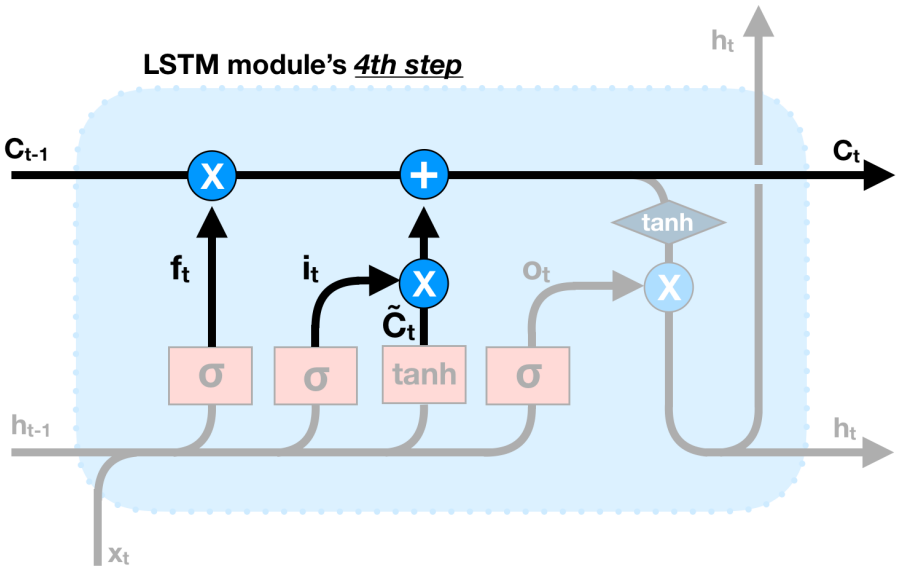


Figure 3.14. Cell state update

The 5th and the last step, illustrated in figure 3.15, is where we generate the prediction output (h_t) of the LSTM module. This prediction will be based on the new cell state (C_t) that was just created in the previous step. To get the prediction output (h_t), the new cell state vector (C_t) needs to first go through a tanh layer, which pushes all the values between -1 and 1. This modified tanh cell state vector (C_t) will be then multiplied with yet another input forget gate called the "output gate "(o_t), which is calculated in equation 3.12, to make sure that the output includes only relevant information. The mathematical notation of generating the hidden state h_t is presented in the equations: 3.13

$$o_t = \sigma(W_o \cdot V_{input} + b_o) \tag{3.12}$$

$$h_t = o_t * tanh(C_t) \tag{3.13}$$

3.5 Evaluating Language Models

There are two ways of evaluating a language model's performance, either extrinsically and intrinsically. Extrinsic evaluation refers to embedding the language model in an application, letting users test the updated sys-

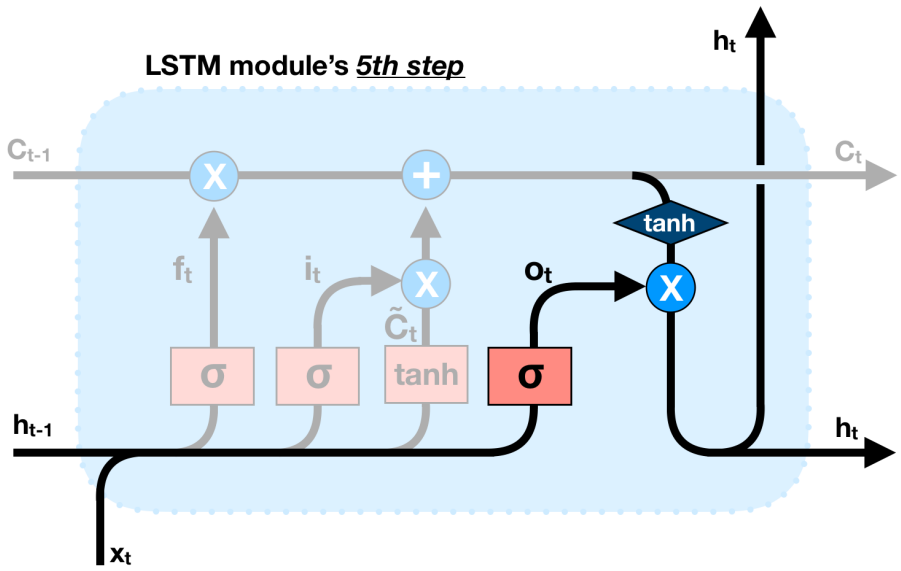


Figure 3.15. Output gate

tem, and telling how much they think the application improves the quality of their end-user experience. For example, an extrinsic evaluation could be performed by embedding a language model to a smartphone's autocompletion tool. With this embedded autocompletion tool, the users could score how useful the autocompletion is and how much time it saves from them or how often they use it.

Intrinsic evaluation metrics make it possible to measure the quality of a model detached from any particular application, which means that it makes it easier to compare with other models (Jurafsky and Martin 2014). In language modeling, the most commonly used intrinsic evaluation metric is perplexity (Goodman 2001; Tomáš Mikolov 2012). For doing an intrinsic evaluation, there needs to be a corpus of data that acts as a test set. This corpus of test data is separate from the training data that is used to train the language model. The test set also ensures that the model will not be over trained with the particular training data, which is called overfitting a model with training data (Jurafsky and Martin 2014).

These two evaluation strategies complement each other. However, this thesis only uses intrinsic evaluation because the goal of the experiment section 6 is to verify that different NNLM toolkits are able to produce comparable models. Hence, intrinsic (perplexity) testing is sufficient and there is no need for extrinsic evaluation as it is vaguer and a much slower method for evaluating language models.

3.5.1 Perplexity

A language model's perplexity (PPL) on a given test set is the inverse probability of the test set, normalized by the number of words (Jurafsky and Martin 2014). When the test set is $W = w_1, w_2..., w_n$:

$$\begin{aligned} PPL(W) &= P(w_1, w_2, ..., w_n)^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P(w_1, w_2, ..., w_n)}} \end{aligned} \tag{3.14}$$

It is possible to use the chain rule for expanding the probability of W:

$$PPL(W) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_1, w_2, ..., w_n)}} \tag{3.15}$$

As an example, for a bigram (or 2-gram) language model this would be computed as follows:

$$PPL(W) = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{i-1})}} \tag{3.16}$$

As this equation shows, the PPL is high when the probability of the words in a particular sequence is low. However, we want to maximize the conditional probability of word sequences and minimize the PPL of a test set. In practice, this means that we are trying to create a language model that can mimic the word sequences of the test set so that it is able to predict a subsequent word when given the previous words. When assuming that the test set is a perfect representation of the language, it means that the smaller the PPL is, the better the language model performs in creating the actual utterances used in the language (Jurafsky and Martin 2014).

Perplexity has useful properties. For instance, PPL can be easily computed for a set of test data. Computing PPL with any language model with a single training and test set makes it ideal for comparing different language models' performances. This fact makes it one of the fastest overall quality metrics when comparing language models. (Jurafsky and Martin 2014; Goodman 2001; Tomáš Mikolov 2012)

Perplexities also have downsides. First of all, an improvement in PPL (intrinsic) does not necessarily mean an extrinsic performance improvement in a language processing application such as machine translation or speech recognition. Hence, LM's improvement should always be tested also extrinsically in the application before concluding the evaluation of the

model. However, the PPL often correlates with the extrinsic improvement of the language model; consequently, it is commonly used as an indicator of the quality of the language model. It is also important to note that the perplexities of two different language models are comparable only if they have the same vocabularies. (Jurafsky and Martin 2014)

3.6 Lexical unit selection for NNLM

Neural network language models can be built in a few different ways when it comes to the training data format. The traditional way of training language models has been to use word-based training data. However, word-based models have some drawbacks, and hence, other techniques have been developed. Other models built to overcome the challenges that word-based models face are sub-word based models, character-based models, and combinations of these two models. This section introduces these different ways of building NNLMs.

3.6.1 Word-based models

Word-based language models have the benefit of being the most intelligible language models. It is easy for a human being to understand how a word-level based language model works when it estimates the probability of a sequence of words. Word-based embeddings are also well suited for capturing the distributional similarity between words. However, there are also disadvantages of using only word-based language models. (Jurafsky and Martin 2014)

One of the disadvantages related to these word-based language models arises when the vocabulary of these models grow. To calculate the probability distribution with a vast vocabulary becomes computationally heavy and slows down the system. The reason for this is the amount of calculated inner products that are the vocabulary size (V) times the word vector (w) length ($len(V) * len(w)$) which in turn radically slows down the updates on the gradient descent (Morin and Bengio 2005). This problem arises with some languages that simply have too vast a lexicon to represent every word as an embedding.

Fortunately, there are methods that seek to address this challenge. Some examples of these are Hierarchical Softmax (Morin and Bengio 2005), Importance Sampling (Bengio, Senécal, et al. 2003), class-based models

(Brown et al. 1992), Noise Contrastive Estimation (Gutmann and Hyvärinen 2010; Mnih and Kavukcuoglu 2013), and self normalizing partition functions (Brébisson and Vincent 2015).

Another drawback is that even languages or applications with manageable lexicons will encounter unknown words due to spelling mistakes and new and borrowed words from other languages (Jurafsky and Martin 2014). These word-based language models can only model the words that they know. They are limited only to the words that have been included in the model initiation phase when the initial word vectors were created. If the language model sees words that it does not know, it will replace that word with an <unk> (unknown) token. Replacing words with <unk> tokens decreases the accuracy and performance of the language models due to the loss of the structure and sense of a sentence.

Having an upper limit for vocabulary is a major problem with agglutinative languages because even some common words might not be included in the "known" corpus. The Finnish language is one such problematic language because it is possible to create words by concatenating morphemes. Hence, it has a vast amount of infrequent words that are some morphological variants, which creates an extensive vocabulary, making word-based models impractical (Kurimo et al. 2006).

3.6.2 Sub-word based models

Solutions based on larger sub-word units have proven to be able to deal with new words and offer reasonable accuracy and training speed (Tomáš Mikolov 2012). Sub-word approaches also have some drawbacks, such as the specification of the sub-word unit creation, which often differs from language to language. Also, the fact that a word can have multiple different segmentations into sub-word units depends on the context (Bojanowski, Joulin, and Tomas Mikolov 2015). For instance, in the Finnish language, the word "kuusi" might mean the number six, spruce, or "your moon". Depending on the context, the ideal sub-word units would be either "kuu" (moon) "-si" (suffix for your) or "kuusi" (six or spruce).

3.6.3 Character-based models

Character-based language models solve problems in capturing the similarity of words like "drink", "drinks", and "drinking", unlike the word-based model (if not using pre-trained word embeddings like glove from Penning-

ton, Socher, and Manning (2014)). Also, character-based models do not need to decide how to spit words as the vocabulary is just all the alphabets. However, they have their drawbacks. For example, to successfully model long-term dependencies, we need large hidden representation, which means higher computational costs, that might become unreasonable in practice (Bojanowski, Joulin, and Tomas Mikolov 2015). One successful approach to overcome some problems of both word-based and character-based language models is a model that uses both of them as an input (Verwimp, Pelemans, Wambacq, et al. 2017; Jurafsky and Martin 2014).

4. MatsuLM

The main goal if this work is to present a simple PyTorch-based NNLM toolkit called MatsuLM[1]. It is built for training NNLMs that can be used for word prediction in autocompletion tools, scoring the "rightness" of sentences, or generating text. The tool has been written using a Python library called Pytorch, which allows developers and researchers to modify neural networks and tune the training process effortlessly. This effortless tuning is essential for both research and industry since the models have to be trained with multiple different configurations to figure out how to develop the best performing language models.

One part of this LM development process includes so-called hyperparameter optimization, where the different default parameters are set for the models and observed which parameters perform the best (Bergstra et al. 2011; Falkner, Klein, and Hutter 2018). The other part of this LM development process is to build different model architectures and compare their results with each other. All of this is made possible with Pytorch, and the new toolkit called MatsuLM. This new toolkit aims to make it easier and faster to run the previously mentioned training and testing comparisons to speed up research and product development.

In addition to the flexibility, Pytorch is optimized to utilize multiple GPU and CPU cores to speed up and parallelize the massive numerical computation. Computational optimization is critical to keep the research and product development iteration cycle as fast and productive as possible. GPU utilization is essential because the training times of neural networks can be 10 to 100 times faster on GPUs than CPUs (Chen et al. 2014; Colic, Kalva, and Furht 2010). Concerning parallel computation, even with the most efficient GPUs, the training time of complex and computationally heavy models can take multiple days (You et al. 2019). Hence, it is crucial

[1] https://github.com/RikoNyberg/matsulm

to parallelize the computation to multiple GPUs. Then, it is still possible to decrease the computation time significantly even if there are no possibilities to improve the computation time by optimizing the code or hardware.

The drawbacks of MatsuLM are that it is only a small tool and does not yet have comprehensive testing in place, and there are only a few integrations build for it. Also, the amount of documentation and example code is more limited than in the existing tools that have been used in a variety of project settings.

4.1 Toolkit description

MatsuLM is currently the most up to date language modeling toolkit created with PyTorch. It is compatible with the latest PyTorch version and created using the latest best practices of machine learning models. MatsuLM is created to be as lightweight as possible. These are the reasons why MatsuLM seems to be faster than other NNLM toolkits. When the models are using the latest PyTorch libraries, they can use modern GPUs as efficiently as possible. MatsuLM is also including all the necessary tools to track and save the models that are being trained.

MatsuLM currently includes an LSTM algorithm. There is a simple LSTM version, which is used in this master's thesis, but there are already built-in options that can be used to create more complex LSTM models. As an example, the LSTM can be bidirectional or the optimizer, and the amount of the layers can be redefined. The complete list of all the editable hyperparameters can be found here:

- num_layers: Number of LSTM layers
- bidirectional: Define if the LSTM will be bidirectional
- embed_size: Word embedding size
- hidden_size: Hidden layer size
- init_scale: Initial scale where the models' weights are uniformly spread (e.g. init_scale=0.5 creates weights between -0.5 and 0.5)
- init_bias: Initial bias of all the models' weights
- dropout: Percentage of the words that will be dropped from each training input and between each LSTM layer if there is more than

one layer.

- weight_decay: A percentage that creates a penalty (L2) to the cost. This should lead to smaller model weights.
- optimizer: Option for choosing an optimization algorithm (e.g. "sgd" or "adam")
- seq_length: Length of one sequence used for the training. Currently, the count of words.
- batch_size: Number of sequences in one training batch
- num_epochs: Number of times the same training data is used for training.
- lr: The learning rate of the LSTM model
- lr_decay_start: The epoch where the learning rate starts to be decay
- lr_decay: Percentage on how much the learning rate will decay after every epoch
- clip_norm: The minimum and maximum weight which will be enforced by clipping a weight if it goes over or under (e.g., clip_norm=5 means that the min is -5 and max is 5)

Other parameters that can be edited are:

- log_interval: Interval of logging the training results
- cuda: Defining if the GPU is used for the model training
- seed: Seed for the language model to make it possible to replicate the results later on
- save_model: Define if the best performing model will be saved
- model_path: Define where the model will be saved

The additional tools used in MatsuLM help in the process of language modeling development and research. These helper tools are making it very easy to do hyperparameter optimization and track what kind of language models have been trained by saving the parameters and the results of the trained models.

MatsuLM includes a simple hyperparameter search tool that has been partially tailored for this toolkit with the use of a so-called flatten-dict

[2] python library. The hyperparameter search works by only listing each wanted parameter to the input dictionary. The hyperparameter search tool will then create all possible combinations of the given parameters for the trained models. It will also take care of saving the best language model versions from each of the hyperparameter combinations.

The tracking and viewing of the models and the training progress are done with an integrated machine learning experiment management tool called Sacred[3]. Sacred is an open-source Python library that makes sure that all of the machine learning experiments and their configurations will be recorded and stored. Storing them ensures that none of the experiments go to waste and that they can be re-created later. To get the most out of Sacred, MatsuLM also includes an instruction for the effortless setup of Omniboard[4], a web dashboard for the Sacred tool.

The use of the hyperparameter search or the Sacred integration is optional but extremely useful. Details on how to set up and use these additional tools with MatsuLM can be found from the GitHub page of MatsuLM[5]. The MatsuLM structure is illustrated in figure 4.1.

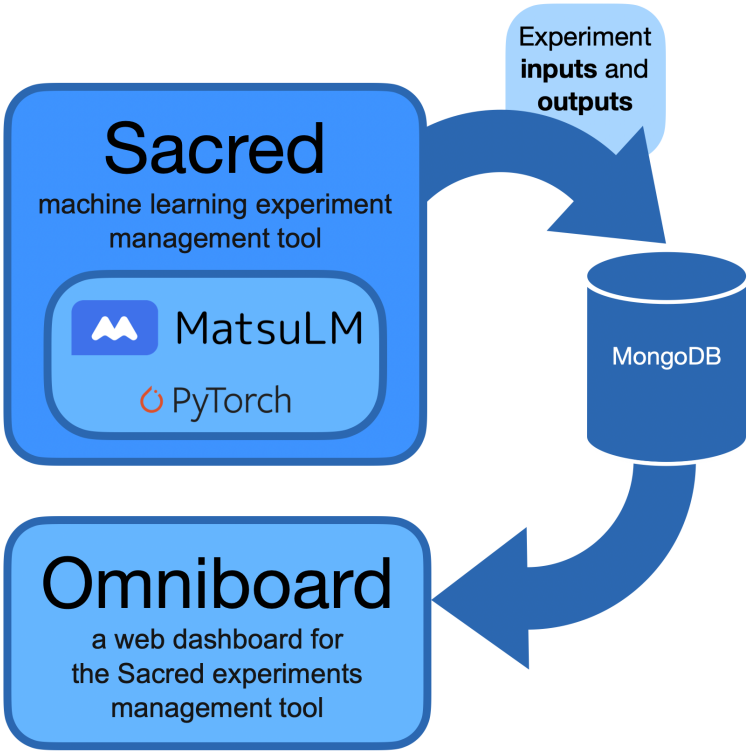


Figure 4.1. Illustrating MatsuLM structure

[2]https://github.com/ianlini/flatten-dict
[3]https://github.com/IDSIA/sacred
[4]https://github.com/vivekratnavel/omniboard
[5]https://github.com/RikoNyberg/matsulm

4.2 Adding new functionalities

Some additional functionalities will be implemented in MatsuLM soon. For example, a possibility to automatically use the pre-trained word2vec or glove word embeddings in the trained language models, support for subwords, and the ability to rescore n-best lists (similarly as in TheanoLM). Anyone is welcome to add these or other new functionalities that would be useful to have in the MatsuLM toolkit. New features will be added to MatsuLM through pull requests in GitHub, and the maintainer merges them. If there are any issues, requests, bugs, or uncertainty on what should be included in the MatsuLM toolkit, please create a new issue to the MatsuLM's GitHub repo.

5. Experimental setup

The experiments conducted during this thesis work utilize three different NNLM toolkits: MatsuLM, TheanoLM, and awd-lstm-lm. MatsuLM is the new NNLM toolkit that this thesis is presenting. TheanoLM and awd-lstm-lm are existing NNLM toolkits that have been built for similar purposes. All of these three toolkits are compared to each other for the purpose of benchmarking the functionalities of each tool.

Additionally, there is one other promising NNLM toolkit that was also examined during the research. This tf-lm NNLM toolkit is described in a paper by Verwimp, Van Hamme, and Wambacq (2019). It is built on top of the TensorFlow library, unlike any of the other examined NNLM toolkits. Unfortunately, this tf-lm toolkit was still under development during the time that this thesis work was ongoing, and the experiments were blocked due to bugs in the toolkit. Hence, this toolkit was excluded from the experiments conducted in this thesis.

Also the datasets and model architectures are described in this chapter. This work used two different datasets to ensure that the results are not dataset dependent and the experiments were done with a LSTM model which is described in section 5.5.

5.1 TheanoLM

ThenoLM[1] is a NNLM toolkit built with a Python library called Theano. Theano allows a user to conveniently customize neural networks, while simultaneously generating effective code that can efficiently use multiple GPUs and CPUs for speeding up the model training by parallelizing computation. TheanoLM adds to this by making it easy to create arbitrary network architectures or to implement new layer types and optimization

[1] https://github.com/senarvi/theanolm

methods. It also includes the popular layer types, like long short-term memory (LSTM) or gated recurrent units (GRU), and optimizers, like AdaGrad or Adam. TheanoLM also has implementations with other tools and libraries, for example, to Kaldi and Morfessor. The TheanoLM toolkit was published with a research paper by Enarvi and Kurimo (2016). This toolkit has been used and also extended by Enarvi, Smit, et al. (2017) and Smit et al. (2017).

However, one major drawback with TheanoLM is that the support and development has been ended for the Theano Python library. Hence, TheanoLM is increasingly missing necessary updates, which makes the software incompatible with some software updates or new hardware, e.g., GPUs. Incompatibilities create unexpected bugs, as this thesis shows, and make the performance of the toolkit unstable and slow compared to other modern NNLM toolkits.

While completing this thesis, the use of TheanoLM software turned out to be one of the most time-consuming processes. To get the TheanoLM working with the latest GPUs (Quadro P5000), it was necessary first to download an old version of this toolkit from Github because the latest version had unresolved issues. Then in order to make the GPU driver work, it also needed a manual update to one of its' python libraries called gpuarray. Also, the training of the models had problems when running more epochs because there was a feature that reran the same epoch with a smaller learning rate if the perplexity or error did not improve enough. This kind of behavior leads to looping indefinitely, which prevented running as many epoch as with the other toolkits.

5.2 awd-lstm-lm (by Salesforce)

The awd-lstm-lm[2] is an NNLM toolkit that is built with the popular Python-based machine learning toolkit Pytotch. This NNLM toolkit has been used in two Salesforce Research papers by Merity, Keskar, and Socher (2017) and Merity, Keskar, and Socher (2018). Awd-lstm-lm is a versatile toolkit that gives easy access to create a different kind of NNLM. Moreover, because it is built on top of the developing and well-supported Pytorch toolkit, the toolkit's performance is up to date. The toolkit is built according to the modern machine learning software standards, making it accessible for any software developer who has used Pytorch.

[2]https://github.com/salesforce/awd-lstm-lm

The drawback that this awd-lstm-lm toolkit has is that it has not been maintained or supported for the last few years. Active maintaining of the toolkit is significant because the speed of the NLP research and the development of machine learning libraries has been faster than ever in the past few years. Hence, this toolkit starts to have bugs and some incompatibilities. For instance, the latest PyTorch version that awd-lstm-lm claims to support is 0.4 when the latest stable PyTorch version is 1.5. Despite this, the awd-lstm-lm is useful as it can be easily updated manually to be compatible with more recent Pytorch versions. However, this updating requires time and might break the toolkit.

5.3 Datasets and preprocessing

The experiment datasets used in this thesis are Penn Treebank (PTB)[3] and Wikitext-2 (wiki-2)[4]. Both of them are widely used datasets for experimenting (Merity, Keskar, and Socher 2017; Yang et al. 2017) and benchmarking language models[5] (Ruder 2020).

The PTB dataset consists of 929k training words, 73k validation words, and 82k test words. In this experiment, we have used the preprocessed version provided by Tomáš Mikolov, Karafiát, et al. (2010). All the words are lowercased, punctuations are removed, and numbers are replaced with the letter N. The vocabulary is the 10000 most frequent words, and all other words are replaced with <unk> (unknown) tokens. There is no additional preprocessing done to this dataset during this research.

The vocabulary size of PTB is relatively small compared to other modern datasets. Hence, this research will also experiment with the wiki-2 dataset Merity, Xiong, et al. (2016). The wiki-2 dataset has a vocabulary size of 33278 words, and it consists of 2 million training words, 217k validation words, and 245k test words. These words have been extracted from the set of verified good and featured articles on Wikipedia[6]. The wiki-2 preprocess replaced out of vocabulary words also with <unk> tokens. All the punctuation marks, numbers, and most of the special characters are left in the text, and hence, it is seen as a more realistic real-life dataset compared to PTB.

[3] http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz
[4] https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-2-v1.zip
[5] http://nlpprogress.com/english/language_modeling.html
[6] https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/

5.4 Model architecture

The experiment is executed in three different NNLM toolkits (MatsuLM, awd-lstm-lm, and TheanoLM) using the same architecture in each of them. This same architecture is used when training both of the training datasets (PTB and wiki-2).

The model architecture has one lstm layer similar to the one described in sections 3.4. Each of the words in the vocabulary were be transformed into a 100-number long vector that were randomly initialized. The output layer's activation function is softmax.

5.5 Models and training details

The hidden states of the LSTM model will be 256 long vectors, and there will not be dropouts anywhere. The initialization of all the weights and biases is zero, and the used optimizer is stochastic gradient descent. Each of the training sequences will be 35 words long, and these training sequences will be batched into 20 sequences per batch, and the learning rate of these batches is set to 1. The gradients have been set max and min values of 5 and -5. Therefore, gradient values will be clipped if they are under -5 or over 5.

Summary of the models training details

```
parameters: {
    "model": {
        "num_layers": 1,     # number of layers
        "embed_size": 100,   # word vector embedding size
        "hidden_size": 256, # hidden state size (inside LSTM)
        "init_scale": 0,     # initial weight scaling
        "dropout": 0,        # dropout
        "init_bias": 0,      # initial bias
        "forget_bias": 0,    # initial forget bias
        "vocab_size": 33278 # vocabulary size
    },
    "cuda": true,        # running on GPU
    "optimizer": "sgd", # optimizer: stochastic gradient descent
    "num_epochs": 20,    # number of epochs
    "lr": 1,             # learning rate
    "seq_length": 35,    # sequence length
```

```
    "batch_size": 20,    # batch size
    "clip_norm": 5       # max and –min gradient (clipping value)
}
```

The training was done on a Linux server with a GPU (Quadro P5000), including 16 gigabytes of memory. Each model was separately trained so that they had the full GPU at their disposal.

6. Results from experiment

Each dataset was trained with every toolkit for 20 epochs. That is the epoch count, where the training time of each toolkit was measured. MatsuLM and awd-lstm-lm were also trained for a second round with 40 epochs to make sure that they performed as expected with higher epoch counts, and to see if there were any differences in the toolkits. TheanoLM was not trained for 40 epochs due to its painfully slow performance and a feature or a bug that blocked it going over a particular epoch. This blocking happened because the perplexity improvement became too small, and TheanoLM tried to rerun the same epoch with different learning rates, which led to looping indefinitely.

The figures 6.1 and 6.2 show the variation of perplexity in each epoch for every toolkit. Table 6.1 lists the training time over 20 and 40 epochs of each of the NNLM toolkits with both of the datasets.

Epochs / Dataset	MatsuLM	awd-lstm-lm	TheanoLM
20 / PTB	0h 2m 47s	0h 4m 5s	0h 44m 48s
40 / PTB	0h 5m 30s	0h 7m 44s	
20 / Wiki-2	0h 15m 13s	0h 21m 53s	4h 51m 48s
40 / Wiki-2	0h 30m 14s	0h 43m 2s	

Table 6.1. The amount of time required by each toolkit for running 20 and 40 epochs of training

Table 6.2 is also listing the test set perplexities after each of the toolkits have trained the LSTM model.

Epochs / Dataset	MatsuLM	awd-lstm-lm	TheanoLM
20 / PTB	150.46	148.43	155.45
40 / PTB	144.04	140.94	
20 / Wiki-2	188.19	179.35	320.83
40 / Wiki-2	177.82	168.99	

Table 6.2. Test set perplexity

Based on the experiments and the subjective experience that the au-

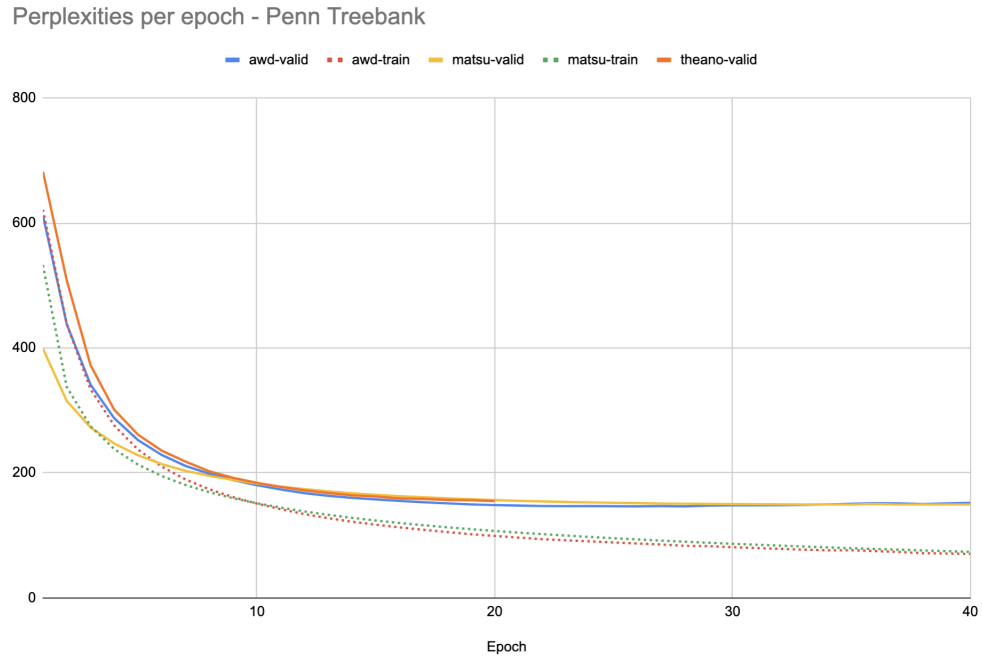


Figure 6.1. NNLM perplexities per epoch with Penn Treebank data - The same LSTM structures in three different toolkits

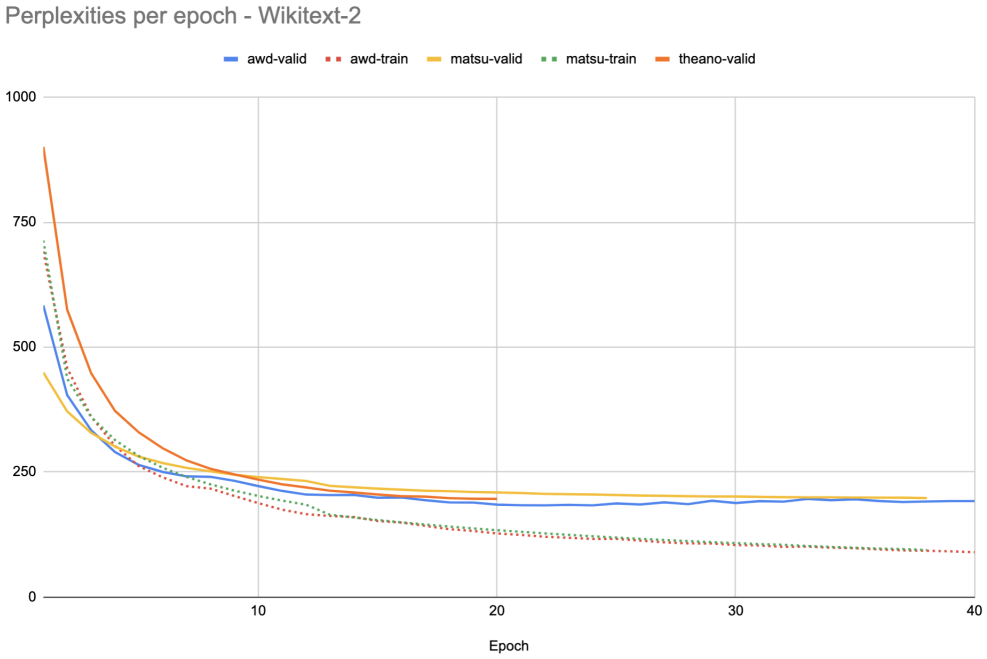


Figure 6.2. NNLM perplexities per epoch with Wikitext-2 data - The same LSTM structures in three different toolkits

thor got from working with each of the NNLM toolkits, it is evident that TheanoLM has become outdated compared to MatsuLM and awd-lstm-lm.

TheanoLM proved to be the most poorly performing and complicated toolkit in all aspects that this thesis experimented with. The similar experience of the toolkit was confirmed also by another software developer who had been using it for research. They mentioned that TheanoLM is quite slow and that the installation is atypical because of the need for installing particular versions of Python libraries Theano and libgpuarray. Only with this particular setup the TheanoLM is able to get the access for GPU.

However, even when accessing the GPU, the most important feature, time used for training of the experimental language models was over 10 times longer than with the newly presented NNLM tool. Some incompatibilities with the new GPUs might be responsible for this slowness, for example, with the efficient memory using and data loading to the GPU, or the slowness may be due to some other unnecessary processes that TheanoLM is performing in the background.

Other complications were setting up the toolkit on a Linux server with a modern GPU. This set-up required manually downloading a specific version, not the latest, of the TheanoLM toolkit and updating its library dependencies to work with the GPU. Running the training had problems because the toolkit started to rerun epochs with lower learning rates when perplexity improvement dropped too low. However, this rerunning of epochs was designed to improve the trained model automatically, so this was an "intentional bug". There were also problems in getting the logging for the training set perplexity, and hence, it was not included in the results chart.

However, the TheanoLM toolkit includes many features and integrations that were not tested in the thesis. Some of these features are unique to TheanoLM and are not included in the other two toolkits. These features and integrations make the TheanoLM toolkit attractive, but the hindrances of using and learning to use it seem to outweigh this attractiveness. Moreover, when also taking into account that the Theano Python library is no longer supported or developed, updating the toolkit is obvious for the Department of Signal Processing and Acoustics at Aalto University.

Between MatsuLM and awd-lstm-lm, the difference in language model training time was not as significant as with the TheanoLM toolkit. The newly presented MatsuLM was about 25 percent faster than awd-lstm-lm

when training the experimental language models. Moreover, the awd-lstm-lm gave slightly lower PPL with the same model structure. This difference in the PPLs might be caused by some internal initialization or automatic optimization of the models within each toolkit or some slightly different way of generating PPL. However, the difference was not significant, and the models were seemingly following the same patters; hence, the differences in PPLs do not seem to be relevant.

The awd-lstm-lm is built on top of the same Pytorch library as MatsuLM but is not compatible with the latest Pytorch version, which might cause some of the sluggishness. Moreover, awd-lstm-lm is also a more complex and comprehensive NNLM toolkit compared to MatsuLM, and may be running some background processes that were unnecessary for the experiments conducted in this thesis.This may also be a logical explanation for the sluggishness of awd-lstm-lm encountered during the experiments.

To summarize this discussion, it can be clearly proven that TheanoLM is the most poorly performing NNLM toolkit, and the newly presented MatsuLM is the most well-performing NNLM toolkit according to the experiments of this thesis. However, MatsuLM is not as extensive as TheanoLM or awd-lstm-lm, nor is it yet as well documented or tested as the other two toolkits. Furthermore, it also does not yet offer a wide range of already implemented algorithms and optimization techniques, as the other two existing toolkits do. This lack of complexity, nonetheless, can also be an advantage when developers or researchers wish to implement something on their own. The purpose of this thesis was not to build an all-inclusive NNLM toolkit but rather a strong foundation for it so that the Aalto University's department or the open-source community can use and develop it further.

7. Conclusion

This thesis focused on making NNLM research faster and easier with a newly presented NNLM toolkit. As a theoretical contribution, this thesis started by investigating the background of language modeling and the hindrances that it faces. To fully understand these, it was also necessary to review the development and working principles of classical and NNLMs. Hence, the thesis examined the related literature and summarized some of the latest achievements and some of the most promising language model structures.

Furthermore, we surveyed the literature and open-source tools for creating NNLMs, to find out what kind of helpful and modern instruments are available. Based on this survey, we found a handful of NNLM toolkits that served these needs. Still, we found out that each of them had issues relating to the outdated and deprecated software or because the tools were not yet appropriately tested and hence, not ready for broader adoption.

This thesis presented a practical contribution to the lack of updated and simple NNLM toolkits by creating MatsuLM, an open-source toolkit for neural network language modeling. It is a modular and straightforward toolkit that makes it easy to modify and continue development as an open-source project.

The comparison experiments that were done for the MatsuLM and existing toolkits had limitations. For example, the data sizes used for these experiments are reasonably small. Hence, we cannot confirm how MatsuLM would perform compared to other existing toolkits when there would be larger training data sizes. However, the comparison results of the smaller datasets indicate them to some extend. Also, the NNLM architectures used for the comparison were simple, which might affect each of the NNLM toolkits' performance in different unexpected ways.

The experiments indicated that this new MatsuLM toolkit outperforms

existing NNLM toolkits with the training speed while performing with similar accuracy. These features will help to speed up the research and development of NNLMs.

8. Future work

The drawbacks of MatsuLM are that it is only a small tool and does not yet have comprehensive testing in place, there is still a limited amount of implemented algorithms, and only a few integrations build for it. Also, the amount of documentation and example code is more limited than in the existing tools that have been used in a variety of project settings. MatsuLM's comparisons to other existing tools are also limited due to the limited time available for this master's thesis. Hence, after adding more algorithms to the MatsuLM, it would be essential to run comparisons to see if it is still training faster than the other NNLM tools.

Hence, the MatsuLM toolkit's future work includes testing with bigger datasets (e.g., Google billion word LM benchmark), more complex NNLM structures (e.g., GRU and transformer models), and building an integration to use existing word embedding models like Glove and Word2vec. Other future work with the toolkit includes adding support for subwords and the ability to rescore n-best lists (as in TheanoLM).

Another important point for future work is to follow the development of one prominent new NNLM toolkit, called tf-lm, that was not included in this master's thesis. This tf-lm toolkit was still under development during the time that this thesis work was ongoing, and hence the comparisons with it were blocked.

Bibliography

Adel, Heike, Katrin Kirchhoff, et al. (2014). “Comparing approaches to convert recurrent neural networks into backoff language models for efficient decoding”. In: *Fifteenth Annual Conference of the International Speech Communication Association*.

Adel, Heike, Ngoc Thang Vu, et al. (2013). “Recurrent neural network language modeling for code switching conversational speech”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, pp. 8411–8415.

Arisoy, Ebru et al. (2012). “Deep neural network language models”. In: *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pp. 20–28.

Bengio, Yoshua, Réjean Ducharme, et al. (2003). “A neural probabilistic language model”. In: *Journal of machine learning research* 3.Feb, pp. 1137–1155.

Bengio, Yoshua, Paolo Frasconi, and Patrice Simard (1993). “The problem of learning long-term dependencies in recurrent networks”. In: *IEEE international conference on neural networks*. IEEE, pp. 1183–1188.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi (1994). “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2, pp. 157–166.

Bengio, Yoshua, Jean-Sébastien Senécal, et al. (2003). “Quick training of probabilistic neural nets by importance sampling.” In: *AISTATS*, pp. 1–9.

Bergstra, James S et al. (2011). “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems*, pp. 2546–2554.

Bojanowski, Piotr, Armand Joulin, and Tomas Mikolov (2015). “Alternative structures for character-level RNNs”. In: *arXiv preprint arXiv:1511.06303*.

Brébisson, Alexandre de and Pascal Vincent (2015). “An exploration of softmax alternatives belonging to the spherical loss family”. In: *arXiv preprint arXiv:1511.05042*.

Brown, Peter F et al. (1992). “Class-based n-gram models of natural language”. In: *Computational linguistics* 18.4, pp. 467–480.

Carbune, Victor et al. (2020). “Fast multi-language lstm-based online handwriting recognition”. In: *International Journal on Document Analysis and Recognition (IJDAR)*, pp. 1–14.

Chelba, Ciprian et al. (2013). “One billion word benchmark for measuring progress in statistical language modeling”. In: *arXiv preprint arXiv:1312.3005*. URL: https://arxiv.org/pdf/1312.3005.pdf.

Chen, X. et al. (2014). “Efficient GPU-based training of recurrent neural network language models using spliced sentence bunch”. In: *INTERSPEECH-2014*, pp. 641–645. URL: https://www.isca-speech.org/archive/interspeech_2014/i14_0641.html.

Colic, Aleksandar, Hari Kalva, and Borko Furht (2010). “Exploring NVIDIA-CUDA for Video Coding”. In: *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. MMSys ’10. Phoenix, Arizona, USA: Association for Computing Machinery, pp. 13–22. ISBN: 9781605589145. DOI: 10.1145/1730836.1730839. URL: https://doi.org/10.1145/1730836.1730839.

Devlin, Jacob et al. (2018). “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805*. URL: https://arxiv.org/pdf/1810.04805.pdf.

Enarvi, Seppo and Mikko Kurimo (2016). “Theanolm-an extensible toolkit for neural network language modeling”. In: *arXiv preprint arXiv:1605.00942*.

Enarvi, Seppo, Peter Smit, et al. (2017). “Automatic speech recognition with very large conversational finnish and estonian vocabularies”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 25.11, pp. 2085–2097.

Falkner, Stefan, Aaron Klein, and Frank Hutter (2018). “BOHB: Robust and efficient hyperparameter optimization at scale”. In: *arXiv preprint arXiv:1807.01774*.

Fan, Yin et al. (2016). “Video-based emotion recognition using CNN-RNN and C3D hybrid networks”. In: *Proceedings of the 18th ACM International Conference on Multimodal Interaction*, pp. 445–450.

Fukushima, Kunihiko (1980). “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological cybernetics* 36.4, pp. 193–202.

Géron, Aurélien (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT press.

Goodman, T Joshua (2001). “A bit of progress in language modeling extended version”. In: *Machine Learning and Applied Statistics Group Microsoft Research. Technical Report, MSR-TR-2001-72*. URL: https://arxiv.org/pdf/cs/0108005.pdf.

Graves, Alex (2013). “Generating sequences with recurrent neural networks”. In: *arXiv preprint arXiv:1308.0850*.

Graves, Alex and Navdeep Jaitly (2014). “Towards end-to-end speech recognition with recurrent neural networks”. In: *International conference on machine learning*, pp. 1764–1772.

Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton (2013). “Speech recognition with deep recurrent neural networks”. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, pp. 6645–6649.

Graves, Alex and Jürgen Schmidhuber (2009). “Offline handwriting recognition with multidimensional recurrent neural networks”. In: *Advances in neural information processing systems*, pp. 545–552.

Gutmann, Michael and Aapo Hyvärinen (2010). “Noise-contrastive estimation: A new estimation principle for unnormalized statistical models”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Jozefowicz, Rafal et al. (2016). “Exploring the limits of language modeling”. In: *arXiv preprint arXiv:1602.02410*.

Jurafsky, Dan and James H Martin (2014). *Speech and language processing. Vol. 3*.

Kannan, Anjuli et al. (2016). “Smart reply: Automated response suggestion for email”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 955–964.

Keselj, Vlado (2019). *Speech and Language Processing Daniel Jurafsky and James H. Martin*.

Kneser, Reinhard and Hermann Ney (1995). "Improved backing-off for m-gram language modeling". In: *1995 International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. IEEE, pp. 181–184.

Krause, Ben et al. (2019). "Dynamic evaluation of transformer language models". In: *arXiv preprint arXiv:1904.08378*.

Kurimo, Mikko et al. (2006). "Unlimited vocabulary speech recognition for agglutinative languages". In: *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*. Association for Computational Linguistics, pp. 487–494.

Lankinen, Matti (2016). "Modeling Finnish language with character-word compositional Language Model". MA thesis. Aalto University.

LeCun, Yann et al. (1999). "Object recognition with gradient-based learning". In: *Shape, contour and grouping in computer vision*. Springer, pp. 319–345.

Lin, Tsungnan et al. (1996). "Learning long-term dependencies in NARX recurrent neural networks". In: *IEEE Transactions on Neural Networks* 7.6, pp. 1329–1338.

Liu, Chuanhe et al. (2018). "Multi-feature based emotion recognition for video clips". In: *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, pp. 630–634.

Merity, Stephen, Nitish Shirish Keskar, and Richard Socher (2017). "Regularizing and optimizing LSTM language models". In: *arXiv preprint arXiv:1708.02182*.

– (2018). "An Analysis of Neural Language Modeling at Multiple Scales". In: *arXiv preprint arXiv:1803.08240*.

Merity, Stephen, Caiming Xiong, et al. (2016). "Pointer sentinel mixture models". In: *arXiv preprint arXiv:1609.07843*.

Mikolov, Tomáš (2012). "Statistical language models based on neural networks". In: *Presentation at Google, Mountain View, 2nd April* 80.

Mikolov, Tomáš, Martin Karafiát, et al. (2010). "Recurrent neural network based language model". In: *Eleventh annual conference of the international speech communication association*, pp. 1045–1048. URL: https://www.isca-speech.org/archive/interspeech_2010/i10_1045.html.

Mikolov, Tomáš, Stefan Kombrink, et al. (2011). "Extensions of recurrent neural network language model". In: *2011 IEEE international conference*

on acoustics, speech and signal processing (ICASSP). IEEE, pp. 5528–5531.

Mikolov, Tomas et al. (2013). “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781*.

Mnih, Andriy and Koray Kavukcuoglu (2013). “Learning word embeddings efficiently with noise-contrastive estimation”. In: *Advances in neural information processing systems*, pp. 2265–2273.

Morin, Frederic and Yoshua Bengio (2005). “Hierarchical probabilistic neural network language model.” In: *Aistats*. Vol. 5. Citeseer, pp. 246–252.

Olah, Christopher (2015). *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014). “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Popkes, Anna-Lena (2018). “Language Modeling with Recurrent Neural Networks - Using Transfer Learning to Perform Radiological Sentence Completion”. MA thesis. Rheinische Friedrich-Wilhelms-Universität Bonn. URL: http://alpopkes.com/files/thesis_APopkes.pdf.

Radford, Alec et al. (2019). “Language models are unsupervised multitask learners”. In: *OpenAI Blog* 1.8, p. 9. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Rong, Xin (2014). “word2vec parameter learning explained”. In: *arXiv preprint arXiv:1411.2738*.

Rosenfeld, Ronald (2000). “Two decades of statistical language modeling: Where do we go from here?” In: *Proceedings of the IEEE* 88.8, pp. 1270–1278.

Ruder, Sebastian (2020). *Language Modeling | NLP-progress*. URL: http://nlpprogress.com/english/language_modeling.html (visited on 05/23/2020).

Sak, Hasim, Andrew W Senior, and Françoise Beaufays (2014). “Long short-term memory recurrent neural network architectures for large scale acoustic modeling”. In:

Smit, Peter et al. (2017). “Aalto system for the 2017 Arabic multi-genre broadcast challenge”. In: *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, pp. 338–345.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). “Sequence to sequence learning with neural networks”. In: *Advances in neural information processing systems*, pp. 3104–3112.

Ullah, Amin et al. (2017). “Action recognition in video sequences using deep bi-directional LSTM with CNN features”. In: *IEEE Access* 6, pp. 1155–1166.

Vaswani, Ashish et al. (2017). “Attention is all you need”. In: *Advances in neural information processing systems*, pp. 5998–6008.

Verwimp, Lyan, H Van Hamme, and Patrick Wambacq (2019). “TF-LM: Tensorflow-based language modeling toolkit”. In: *LREC 2018-11th International Conference on Language Resources and Evaluation*. Proceedings LREC, pp. 2968–2973.

Verwimp, Lyan, Joris Pelemans, Patrick Wambacq, et al. (2017). “Character-word lstm language models”. In: *arXiv preprint arXiv:1704.02813*.

Yang, Zhilin et al. (2017). “Breaking the softmax bottleneck: A high-rank RNN language model”. In: *arXiv preprint arXiv:1711.03953*.

You, Yang et al. (2019). “Large batch optimization for deep learning: Training bert in 76 minutes”. In: *International Conference on Learning Representations*. URL: https://arxiv.org/abs/1904.00962.