Abstract

The research question of this thesis is to answer whether it is possible to generate text on mobile devices. Therefore, we present an approach to constructing text generation language models and embedding them into mobile applications. The result of such embedding is lower latency and improved end-user experience, as no Internet connection is required. This approach can also enhance privacy, as user data is stored only on the user's mobile device, and can produce significant savings, as no server is required to run these models.

Contents

Chapter 1

Introduction

1.1 Motivation

Text generation, also referred to as "Text Prediction" or "Language Modeling" in the scientific literature [Gra13, p. 6], is a very active area of research, particularly after recent breakthroughs in neural language modeling.

Predicting the next word or character in a sentence, capturing syntactic rules and semantics to create cohesive text is fascinating. Language models can be applied not only to natural languages but also to any text data, including computer code. When text generation reaches human results, an event that might happen in the near future, its impact will be profound, particularly in the areas of communication and content creation.

The main focus of this thesis is to bring language modeling to mobile applications in order to generate text locally on the device.

In a production environment, machine learning models typically run on dedicated servers, and their outputs are consumed by web browsers, mobile devices, and embedded systems.

However, machine learning algorithms can be executed on mobile devices, a process known as “on-device inference.” Recently, there has been an explosion in its implementation in applications such as Snapchat, Prisma and many others.

Some of the advantages are:

- On-device inference. Since the models are built into client applications, they do not require an Internet connection, resulting in low latency.
- Privacy. Users do not disclose their personal information because all the data they enter is retained on the device.
- Expense reduction. Inference occurs on the device, which means there are no server costs. The only resource consumed is the device’s battery.

- Security. No data is transferred over the network, increasing overall safety.

Taking into account the advantages mentioned above, both for users and service providers, it makes sense to study this topic in more detail.

It is important to note that the approach presented in this thesis can be applied to a number of tasks. Any sequence prediction task, such as machine translation and text summarization, can be performed on a mobile device in a very similar way.

1.2 Project Objective

The project objective is to develop a text generation model and embed it in a mobile application for on-device inference.

Since the capacity of mobile devices is not comparable to the capacity of servers, the model must be small enough to be executable within the restrictions of the mobile device. Due to a large number of calculations performed during inference, large models are likely to be unreliable in a production environment. Excessive use of memory will also quickly lead to the impossibility of deployment.

Therefore, the goal of the project does not include the achievement of current state-of-the-art results, as it is simply impossible to implement large models on mobile devices. The resulting mobile application will be available only for iOS or Android devices.

1.3 Proceeding and Structure of the Work

The remainder of this thesis is structured as follows:

- Chapter 2 explains both the theoretical part of the work and the available technical tools. The datasets are also discussed in this chapter.
- Chapter 3 presents a project analysis, including functional and non-functional requirements.
- In Chapter 4, the tools, language models and the user interface are analyzed and chosen.
- In Chapter 5, the implementation is realized.
- In Chapter 6, the evaluation is performed.
- Chapter 7 concludes the thesis and lists possible future work.

Chapter 2

Fundamentals

2.1 Language Modeling

Language modeling is usually framed as unsupervised distribution estimation from a set of examples $(x_1, x_2, ..., xn)$ each composed of variable length sequences of symbols $(s_1, s_2, ..., sn)$. [Rad19, p. 2]

Therefore, language modeling can be described as follows:

$$p(x) = \prod_{i=1}^{n} p(s_n|s_1, ...s_{n-1}) \tag{2.1}$$

In short, language models show the probability distribution over sequences of words or characters. [HS16, p. 1]

Traditionally used methods are n-gram and neural language models. N-gram models make Markov assumption by estimating n-gram probabilities via counting and subsequent smoothing. [Kim15, p. 1] This thesis focuses on neural language models for several reasons, the main ones being the superior performance and the possibility of deployment to mobile devices.

2.2 Neural language models

2.2.1 Neural network

Neural language models are based on artificial neural networks. D. Jurafsky defines a neural network as a network of small computing units, each of which takes a vector of input values and produces a single output value. [JM18, p. 131]

2.2.2 Neuron

At the core of a neural network is a neuron (unit). The computations inside the classic neuron are defined as follows:

$$z = w \cdot x + b \tag{2.2}$$

where w and x are weight and input vectors, and b is a bias term. Usually, a non-linear function (activation function) must be applied to the output z. For example, a sigmoid function:

$$y = g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{2.3}$$

Neural networks can have several hidden layers, each of which consists of several neurons, leading to simultaneous multiplication between input and weight matrices, which gives a certain output (forward pass).

2.2.3 Feed-forward neural network

A classic example is a feed-forward neural network. It is a multilayer network in which the units are connected with no cycles; the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. [JM18, p. 137]

In the case of the classification problem, we need a vector of probabilities, not a vector with real numbers. We receive it by applying the softmax function, which is done in the output layer:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \le i \le d \tag{2.4}$$

where z is a vector of dimensionality d. [JM18, p. 139] If we sum up all the probabilities after applying this function, the result will be 1.0, which corresponds to 100% probability.

Therefore, a feed-forward neural network can be represented as follows [JM18, p. 140]:

$$\begin{aligned} z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\ a^{[1]} &= g^{[1]}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= g^{[2]}(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned} \tag{2.5}$$

where $a^{[0]}$ is an input to the network, $g^{[1]}$ is an activation function of the hidden layer, and $g^{[2]}$ is a softmax function of the output layer.

2.2.4 Training

Before actual use, a neural network is trained by learning the optimal values of parameters: weights W and biases b. [JM18, p. 140] This is achieved by using a gradient optimization algorithm, such as batch gradient descent:

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{2.6}$$

where θ - parameters, η - learning rate, and the rest is the gradient of the loss function. The learning rate influences the value by which the parameters are updated. The loss function measures how well model performs and determines the gradients from the final layer of the network.

For classification problems the loss function can be represented as negative log likelihood [JM18, p. 141]:

$$L_{ce}(\hat{y}, y) = -log\hat{y}_i \tag{2.7}$$

The gradient of the loss function is a vector containing the partial derivative of the loss function with respect to each parameter. This gradient is calculated using a backpropagation algorithm. In short, it computes the derivative of the loss function L with respect to the weights V. However, since the loss is not expressed directly in terms of the weights, it applies the chain rule to get there indirectly [JM18, p. 181]:

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial V} \tag{2.8}$$

where a is the activation of the output layer, and z is the intermediate network activation. The last term is the activation value of the hidden layer.

2.2.5 Embeddings

Word embeddings exhibit the property whereby semantically close words are likewise close in the induced vector space. [Kim15, p. 1] They can be pre-computed from large datasets and then used during training. Word embeddings allow neural networks to capture the semantics of the text and improve generalization.

With word embeddings, a neural network language model with one hidden layer can be represented as follows [JM18, p. 148]:

$$\begin{aligned} e &= (E_{x_1}, E_{x_2}, ..., E_x) \\ h &= \sigma(W_e + b) \\ z &= Uh \\ y &= softmax(z) \end{aligned} \tag{2.9}$$

where x_i is a one-hot vector of an input, E is an embedding matrix, σ is an activation function, and W and U are weight matrices for hidden and output layers respectively.

Character embeddings are similar to word embeddings, the difference being that these embeddings exist on the character level. They capture morphological and shape information. [SG15, p. 2]

2.2.6 Popular models in language modeling

Currently, the most common neural network architectures for language modeling are recurrent and attention-based. Although recent developments show that attention-based architectures such as Transformer tend to perform very well [Rad19], recurrent neural networks are still popular choice for language modeling, as we can see from current state-of-the-art models. [Rud19]

2.3 Recurrent neural language models

2.3.1 Recurrent neural networks

Jeffrey L. Elman proposed recurrent neural networks (RNNs) in 1990. The central idea is that the recurrent connections allow the network's hidden units to see its own previous output, so that the subsequent behavior can be shaped by previous responses. These recurrent connections are what give the network memory. [Elm90, p. 182]

Recurrent neural networks based language models employ the chain rule to model joint probabilities over word sequences:

$$p(w_1, ..., w_N) = \prod_{i=1}^{N} p(w_i | w_1, ...w_{i-1}) \tag{2.10}$$

where the context of all previous words is encoded. [Joz16, p. 3] This also applies to character sequences.

As explained in [Kim15, p. 2], at each time step t, an RNN takes the input vector $x_t \in \mathbb{R}$ and the hidden state vector $h_{t-1} \in \mathbb{R}$ and produces the next hidden state h_t by applying the following recursive operation:

$$h_t = f(Wx_t + Uh_{t-1} + b) \tag{2.11}$$

where W and U are weight matrices for input and state values, b is a bias term, and f is an activation function. The output is produced as follows:

$$y_t = g(Vh_t + b) \tag{2.12}$$

where V is a third weight matrix and g is an activation function.

2.3.2 Backpropagation through time

Similar to a feed-forward neural network, an RNN uses one of the algorithms of gradient optimization to update its parameters in order to minimize the total loss with respect to the target outputs.

Backpropagation algorithm, which is used in RNNs, is known as backpropagation through time (BPTT). It decomposes the gradient as a sum, over timesteps t, of the effect of a change of parameter at time t on all subsequent losses. The extension of BPPT is Truncated BPTT. It truncates gradient flows after a fixed number of timesteps, or equivalently, splits the input sequence into subsequences of fixed length, and only backpropagating through those subsequences. [TO17, pp. 3-4]

2.3.3 LSTM

One of the most popular variations of RNN is Long Short-Term Memory networks or LSTM. [HS97] They introduce special gates in neural units in order to forget the old state of the network. And instead of manually deciding when to clear the state, a neural network learns to decide when to do it. [GBC16, p. 404]

As described in [Kim15, p. 2], LSTM networks address the problem of learning long range dependencies by augmenting the RNN with a memory cell vector $c_t \in \mathbb{R}$ at each time step. An LSTM takes as input x_t, h_{t-1}, c_{t-1} and produces h_t, c_t via the following intermediate calculations:

$$
\begin{aligned}
i_t &= \sigma(W^i x_t + U^i h_{t-1} + b^i) \\
f_t &= \sigma(W^f x_t + U^f h_{t-1} + b^f) \\
o_t &= \sigma(W^o x_t + U^o h_{t-1} + b^o) \\
g_t &= \tanh(W^g x_t + U^g h_{t-1} + b^g) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
\tag{2.13}
$$

where $\sigma(\cdot)$ and $\tanh(\cdot)$ are the element-wise sigmoid and hyperbolic tangent functions, $\odot$ is the element-wise multiplication operator, and i_t, f_t, o_t are referred to as input, forget, and output gates. At $t = 1$, h_0 and c_0 are initialized to zero vectors.

2.3.4 Stacked RNN and Bidirectional RNN

The widely used recurrent architecture is stacked or deep RNN, which consist of several hidden layers. Another type is bidirectional RNN, which combines an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence. [GBC16, p. 388]

Unfortunately, stacked and bidirectional RNNs significantly increase the number of parameters in the network. This can be a problem when deploying RNNs to mobile devices, for example.

2.3.5 Dropout and Batch Normalization

During training, it is important to avoid overfitting. Overfitting occurs when a neural network achieves good results on its training dataset, but underperforms on the test set. In that case, some regularisation techniques are used. One of the most popular methods is dropout [Sri14], which randomly sets some neurons along with their connections to zero, thus disabling affected neurons from learning.

Another popular way of improving networks' performance is batch normalization. [IS15] During training, the data is usually consumed in batches, but they are not normalized, even if the entire dataset is. By normalizing the batches, we reduce the shifting of neuron values (known as a covariance shift). As a result, the network will train faster and generalize better.

2.4 Word-level and character-level models

When considering a neural network for language modeling, it is essential to choose which input will be consumed by the network. Usually, these are words or sub-word elements, such as characters. It also determines how many classes an output will have. It might be the number of unique words in a vocabulary, or the number of characters.

2.4.1 Word-level models

In most cases text prediction is performed at the word-level. [Gra13, p. 6] Character-level models (CLMs) usually perform much worse than the word-level language models (WLMs). [HS16, p. 1]

WLMs, however, have other shortcomings. They suffer from increased computational cost due to large vocabulary sizes [MKS18, p. 1], which corresponds to the number of unique words in a text corpus. There is also the need to hold huge integer-word mappings of vocabulary in memory because models usually only accept vectors with numeric values.

The second problem of WLMs is a complete inability to sensibly assign a nonzero probability to previously unseen words [Mik12, p. 1], as well as difficulties working with non-word strings, such as web-addresses. [Gra13, p. 6]

The third and biggest problem is the number of parameters. Due to the size of vocabulary, the number of parameters in the word-level model is significantly larger. For example, in [Gra13, p. 7], the number of parameters of WLM is 12 times bigger than of CLM.

2.4.2 Character-level models

Character-level models have shorter vocabularies (i.e., only ASCII characters), can take as an input and produce out-of-vocabulary words, and have much fewer parameters. They also capture grammatical rules of text since punctuation marks are treated as characters.

The downsides of CLMs are lower accuracy and the necessity to make more predictions to achieve a result of the same size as WLMs. Character-level models can also make use of character-level embeddings (as opposed to word-level embeddings).

2.4.3 Byte pair encoding

Another option is to use byte pair encoding (BPE). [SHB15] BPE is a practical middle ground between character and word level language modeling which effectively interpolates between word level inputs for frequent symbol sequences and character level inputs for infrequent symbol sequences. [Rad19, p. 4]

2.5 Perplexity and bits per character metrics

Perplexity is the usual performance measure for language modeling. [Gra13, p. 8] It is the average per-word log-probability on the holdout dataset [Joz16, p. 5]:

$$e^{-\frac{1}{N}\sum_i \ln p_{w_i}} \tag{2.14}$$

The base e can be substituted with 2:

$$2^{-\frac{1}{N}\sum_i^N \log_2 p_{w_i}} \tag{2.15}$$

Character-level models employ a metric called *bits per character* (BPC), which is the average value of $-\log_2 Pr(x_{t+1}|y_t)$. [Gra13, p. 8]

As pointed out by A. Graves [Gra13, p. 8], it is possible to convert BPC of character-level models to *word-level perplexity*: 2^{nBPC}, where n is the average word length on the test set. However, this metric sometimes gives non-comparable results.

Many researchers prefer to use *bits per character* to evaluate CLMs ([HB19, p. 4], [BZ19, p. 2], and *perplexity* for WLMs. If we take a look at one of the largest repositories that track state-of-the-art models in language modeling [Rud19], we can see that CLMs evaluation is usually done with the *bits per character* metric.

2.6 Mobile operating systems

There are two leading mobile operating systems: Android and iOS, maintained by companies Google and Apple, respectively.

Android and iOS are used on a wide variety of mobile devices, including smartphones and tablets.

Currently, neither iOS [19c] nor Android [19b] offer out-of-the-box language modeling for on-device inference, but they do support embedding of custom neural network models developed with machine learning frameworks.

2.7 Machine learning frameworks and tools

2.7.1 Android

The following frameworks and tools can be leveraged to build and use custom neural network models for on-device inference running the mobile Android operating system:

- TensorFlow is an open-source library for developing and training ML models. [19p]
- TensorFlow Lite [19s] is an open-source deep learning framework for on-device inference. The machine learning model must be converted to the TensorFlow Lite format, which can be achieved with the tool called TensorFlow Lite Converter. [19r]
- tf.keras is TensorFlow's implementation of the Keras API specification. [19q]

2.7.2 iOS

The following frameworks and tools can be leveraged to build and use custom neural network models for on-device inference running the mobile iOS operating system:

- CoreML is a framework for integrating machine learning models into iOS applications. [19d] A machine learning model has to be in CoreML format, which can be achieved either by creating a model using CreateML [19f] or by creating a model with other frameworks and then converting it to CoreML format. Currently, CreateML does not support any type of language modeling, so it is not considered.
- Coremltools is a Python package that can be used to convert trained models from popular machine learning tools (including Keras, Caffe, Scikit-learn, libsvm and XGBoost) into CoreML format. [19e]
- Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. Keras supports both convolutional networks and recurrent networks. [19i]

2.7.3 Google Colaboratory

In addition to these frameworks and tools, there is a need for an environment in which machine learning models can be trained.

Google Colaboratory is a research tool for machine learning education and research. It is a Jupyter notebook environment and is free to use. [19g]

One of the main advantages of Google Colaboratory is the ability to use Google's GPUs or TPUs in one remote virtual environment. Training can also be carried out locally.

2.8 Programming languages

2.8.1 Python

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. [19n]

All major machine learning frameworks and tools are written in Python or support it.

2.8.2 Swift

Swift is a general-purpose programming language. Open-source Swift can be used on the Mac to target all of the Apple platforms: iOS, macOS, watchOS, and tvOS. [19a]

Currently Swift is the main programming language for writing mobile application on iOS platform.

2.8.3 Kotlin

Kotlin is an open-source, statically-typed programming language that supports both object-oriented and functional programming. Kotlin provides similar syntax and concepts from other languages, including C#, Java, and Scala, among many others. [19k]

Currently Kotlin is the main programming language for writing mobile application on Android platform.

2.9 Datasets

2.9.1 Works of W. Shakespeare

Martin Görner's collection of William Shakespeare's works is publicly available on his GitHub repository. [Gör17] It contains approximately 960,000 words. This collection is a part of his presentation "Tensorflow and deep learning - without a PhD".

While rarely used in academic research, using Shakespeare works for training RNN language models was popularized by A. Karpathy, in one of his blog-posts. [Kar15]

The dataset will allow us to see if our models can capture Shakespeare's writing style.

2.9.2 Penn TreeBank

This dataset [MSM93] is one of the most popular text datasets used in academic research for language modeling, including word-level and character-level models. It is a collection of Wall Street Journal articles. It contains approximately 1,080,000 words.

The preprocessed version of the dataset is publicly available on Tomas Mikolov's webpage. [Mik10]

This dataset will allow us to compare the perfomance of our models with the current state-of-the-art models.

Chapter 3

Analysis

In this chapter, the requirements of the final application will be established. That includes both functional and non-functional requirements.

3.1 Requirements analysis

First, we will formulate a possible usage scenario, on which functional and non-functional requirements will be based:

- The user opens the mobile application.
- The user provides some text input in English.
- After a reasonable period of time, the mobile application will display the generated text.
- The user can then enter another text input

Based on the created usage scenario, it is possible to establish functional and non-functional requirements.

The functional requirements:

- The application accepts user input in English.
- A user input can be of any length, but not shorter than 5 characters.
- After receiving the input from the user, the application generates a text.
- The generated text should be similar to the text data used to train the model.

The non-functional requirements:

- Efficiency and performance: on-device inference should happen within a reasonable period of time expected from such an application by the user.
- Usability: the application should be intuitive to the user.
- Reliability and safety: personal data should not be persisted.
- Extensibility: there should be a possibility of further application development.

3.2 Research on similar products

Currently, there are several mobile applications on the market that use neural language models. Unfortunately, most owners do not provide any information about them.

One example is the SwiftKey keyboard app. [19o] This mobile application provides text suggestions given user text input. It initially used n-gram models, but with time the developers switched to neural language models, as noted in the official blog of the company. [15] However, they do not provide sufficient details about the neural language model used.

Another application is named Continuous. [19h] This iPad app is an integrated development environment for C# and F# programming languages. As a part of its functionality, it provides code suggestions using the LSTM neural network, as noted in the creator's blog. [Kru18]

Though source code is unavailable, the creator gives some general characteristics of the model. It contains one hidden LSTM layer and one fully-connected (dense) output layer. The number of parameters is 47,348. The model works at word-level (token-level) with a vocabulary size of 150 tokens. The model makes its prediction based on the previous eight tokens. The size of the model is about 200 KB.

The following advantages and disadvantages are based on the mentioned blog post.

The advantages of the model:

- The model is lightweight and has few parameters, which is beneficial for performance.

The disadvantages of the model:

- The range of possible use of the application is very limited. Due to the small vocabulary, the model is executable on a mobile device at the word-level. However, this model is not scalable to texts containing the actual language, such as English.
- No standard model metrics for word-level models (for example, *perplexity*) are shown.

Chapter 4

Conception

This chapter explains the concept of a mobile application with an embedded neural language model that will be developed as a part of this thesis.

4.1 Mobile operating system and tools

As already mentioned, there are two mobile operating systems: iOS and Android. Both of them provide the possibility of embedding custom neural network models but do not offer out-of-the-box language modeling. There is also no detailed documentation on this matter provided.

Therefore, before implementation, it is important to consider the possibility of embedding the model in a mobile application. Since there is no clear API available, this issue should be addressed first. We need to choose a mobile operating system that will allow us to use a language model on a mobile device.

4.2 Android

Currently, it is possible, but problematic to embed a neural language model into Android applications.

For example, it is technically very challenging to convert created models into a format that is supported by Android applications (.tflite format). The official TensorFlow Lite Converter (previously known as "toco") [19r] is highly unstable, especially for RNN conversions, as evidenced by several reported issues in the official TensorFlow repository (examples: [18a], [18b]).

In addition, while certain projects that make use of RNNs on Android do exist, the repositories of these projects have not been maintained for several years (examples: [Cao17], [Qiu16]). In general, due to these issues and the lack of official documentation on this topic, it makes sense to turn to another mobile operating system - iOS.

4.3 iOS

As with Android, there is no official, detailed guide or project available on how to create, convert, and embed language models in iOS applications. However, Apple's official documentation clearly states that it is possible to integrate recurrent neural networks into iOS or macOS applications. [19m]

Therefore, in this thesis models will be created with the help of tools that will allow us to convert and embed such models in iOS-applications.

The models will be built and trained using the Keras framework with the TensorFlow backend in a Google Colaboratory environment, converted to the .mlmodel format with the help of Coremltools Python package, and embedded and used in an iOS application with the help of CoreML framework.

4.4 Preprocessing

For the model to accept the input data, preprocessing must be performed. This includes cleaning up possible impurities, if present, separating the data into training, validation and test sets, and storing the data in an appropriate format. The preferred result is a .txt file containing the necessary text data.

4.5 Language models

In the absence of detailed official documentation on this topic, it is proposed that more than one language model to be developed and evaluated. Since the models will be used in a mobile application that has far fewer resources than a server or even a laptop, it is vital to consider the type of neural network, level, and size of the models.

4.5.1 Type of neural network

As was already mentioned, the most popular neural language models are recurrent neural networks such as stacked LSTM networks and attention-based networks such as Transformer.

This thesis does not take into account the attention-based language modeling, as it is not known whether such models can be embedded in a mobile application. In addition, Keras and CoreML currently do not support these types of models.

Therefore, the type of neural network is going to be recurrent, particularly, it is going to be LSTM.

4.5.2 Level of the models

When considering a language model for a mobile application, we need to decide at what level this model will work. There are three options: character-level models, word-level models, and byte pair models. Of these, the first two are the most popular.

This thesis focuses on the development of character-level models for the following reasons:

- There are much fewer parameters. Parameters play an essential role in defining the actual size of the model that will be deployed to a mobile application. When the input is a vector of integers, the size of weight matrices has to correspond to the size of that vector. If the vocabulary contains 10,000 unique words, then the input vector's size would be 10 000 x 1, and the weight matrix's size would be n x 10,000, where n is the number of neurons in the first hidden layer. Therefore, the smaller the vocabulary, the fewer the parameters.
- Because of the smaller vocabulary, there is no need to hold huge integer-word mappings in the memory of a mobile device.

The disadvantage of a character-level model, in addition to being less accurate, is the possible increased on-device inference, since it must make more predictions in order to have the same size of output as a word-level model. For example, a character-level model should make five predictions to make the word "night", and a word-level model should make only one prediction. However, because of the smaller vocabulary, there are fewer matrix multiplications which presumably will negate this difference.

For the above-mentioned reasons, the models will be developed as character-level models.

4.5.3 Size of the models

If we take a look at the current state-of-the-art in language modeling, most of the models have tens to hundreds of million parameters. [Rud19]

If we take an example of the Continuous app we discussed, the size of the model that is actually used in a successful production mobile application is 200 KB or less, which corresponds to 47,348 parameters. We can assume that the time for on-device inference of one prediction is under 0.1 second.

This means that if we had a model with 10 million parameters (211 times larger), the model would be about 40 MB in size and the time of on-device inference for just one prediction would be more than 20 seconds. This makes the model unsuitable for use in a production environment.

Of course, these are estimated metrics which may not correspond to the actual results, but it is important to limit ourselves to a number of parameters that may be suitable for a mobile application before choosing an actual model for training.

Therefore, a range of 50,000 to 3,500,000 parameters is proposed, which is expected to result in a model size from 200KB to 12MB, and inference time for one prediction of 0.1 to 7 seconds.

4.5.4 Actual models

As was mentioned, the Keras framework will be used to build and train models. It is necessary to use Keras because its models are supported for CoreML conversions, and also because it supports recurrent neural networks.

The reader needs to know that the official Keras framework will be used, and not the tf.keras module, which is TensorFlow's implementation of the Keras API specification. The tf.keras module has much richer functionality, but its models are not supported for CoreML conversion.

Keras imposes some limits to the type of network it can build. It supports recurrent neural networks, but its functionality is limited. For example, it is not possible to create state-of-the-art LSTM networks, such as Fast-Slow RNN. [MMS17] It is also not possible to fine-tune some regularizations.

Because we limit the number of parameters to 3,500,000 and use the high-level Keras API, the networks developed in this thesis do not attempt to mirror, or even approach, state-of-the-art models. Bidirectional RNNs are excluded from consideration because using even one bidirectional layer might double the number of parameters.

Before implementing the mentioned models, we have to consider the length of the input, or how many characters the network will take into account before making a prediction. A standard number among character-level models is 40 characters, which will be used here as well.

The models which will be developed are presented below. Number 40 corresponds to the length of the text string that the model accepts. Number 54 is the size of the vocabulary of Shakespeare's works (the number of unique characters). "None" means that the shape can be arbitrary (depends on the input).

The first model and the second model use a 3-dimensional tensor as an input, where the second dimension corresponds to the timestep of the sequence (size 40), and the last dimension is a one-hot tensor with the size of vocabulary (size 54). The third model uses a two-dimensional vector with the size of 40, where value is the number of the character which corresponds to the integer-character mapping of the vocabulary.

1. A recurrent neural network with LSTM layer which has 128 units. The output layer is a fully-connected dense layer with a softmax activation function. See figure 4.1.

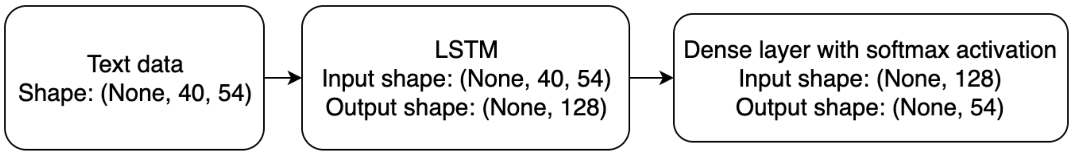

Text data
Shape: (None, 40, 54)
LSTM
Input shape: (None, 40, 54)
Output shape: (None, 128)
Dense layer with softmax activation
Input shape: (None, 128)
Output shape: (None, 54)

Figure 4.1: First model

   This model bears some similarities to the one that was used in the app Continuous as only one hidden layer is used. This type of small model is also featured as an example for text generation in the official Keras repository. [19j] Our model will use different optimizer and set of hyperparameters. It was done to achieve better convergence and avoid possible NaN values during training which we encountered while exploring this model (which could happen due to the gradient explosion, the situation where gradients become very large).

2. A stacked recurrent neural network with two hidden LSTM layers, each of which has 512 units. The model has a dense layer with a softmax activation as the last layer. We use only two LSTM layers because we don't want to increase the limit of 3,500,000 parameters we set. Because this model is larger than the previous one, we apply 0.5 dropout to the second LSTM layer to avoid overfitting. See figure 4.2.

3. A recurrent neural network with character embeddings as its first layer. The second layer is an LSTM layer which has 256 units. The mentioned LSTM layer is followed by two fully-connected dense layers. Batch normalization, as well as ReLU activation [NH10], which is a popular activation function for hidden fully-connected layers, are applied to these layers. The output layer is a fully-connected dense layer with a softmax activation. See figure 4.3.

   This type of model was created by Max Woolf and is publicly available in his repository. [Woo] His model is unique in that it uses character embeddings created

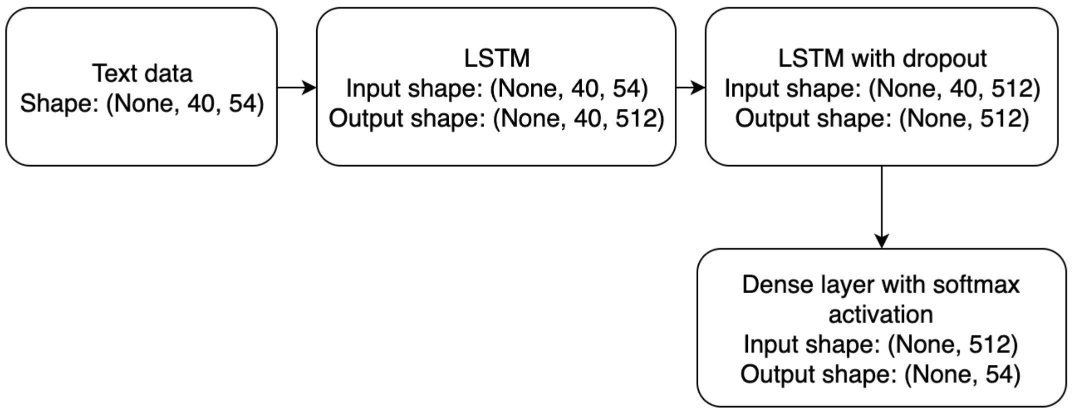

Text data
Shape: (None, 40, 54)
LSTM
Input shape: (None, 40, 54)
Output shape: (None, 40, 512)
LSTM with dropout
Input shape: (None, 40, 512)
Output shape: (None, 512)
Dense layer with softmax activation
Input shape: (None, 512)
Output shape: (None, 54)

Figure 4.2: Second model

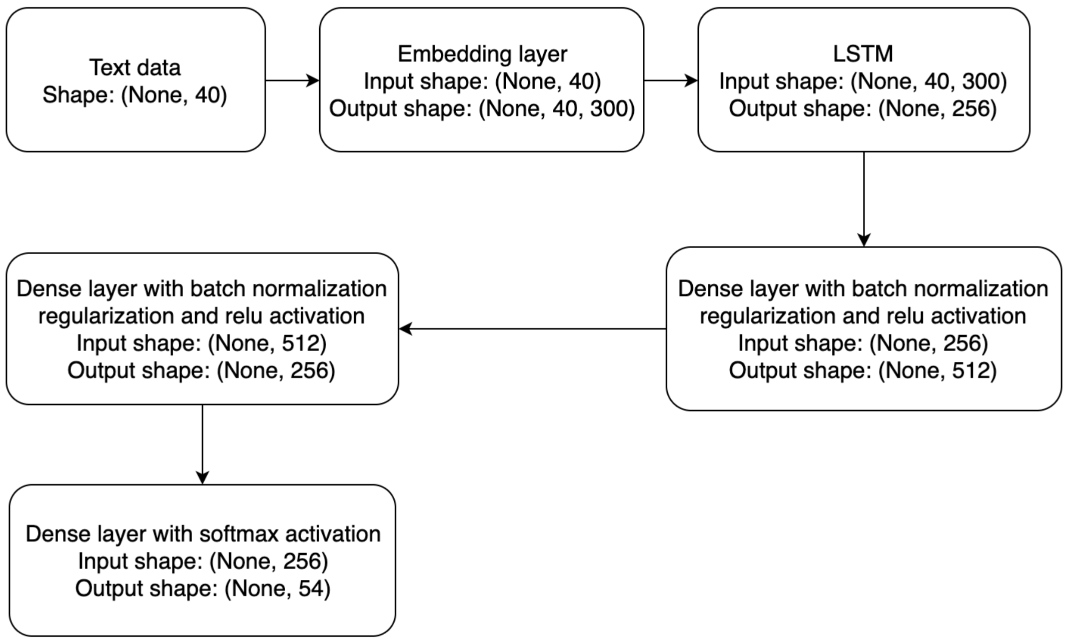

Text data
Shape: (None, 40)
Embedding layer
Input shape: (None, 40)
Output shape: (None, 40, 300)
LSTM
Input shape: (None, 40, 300)
Output shape: (None, 256)
Dense layer with batch normalization regularization and relu activation
Input shape: (None, 256)
Output shape: (None, 512)
Dense layer with batch normalization regularization and relu activation
Input shape: (None, 512)
Output shape: (None, 256)
Dense layer with softmax activation
Input shape: (None, 256)
Output shape: (None, 54)

Figure 4.3: Third model

from pre-trained GloVe word vectors [PSM14] to capture the semantics of each character. In our model, we use a different learning rate and avoid learning rate decay (automatic reduction of learning rate) in order to achieve better convergence.

All models will have Adam as their optimization algorithm. [KB14] Adam is the prevalent choice in the current state-of-the-art character-level LSTM models, for example [MKS18] and [MMS17]. The learning rate will be set to 0.0002, which we can manually decrease once the loss value stop improving. There is no learning rate decay, as according to popular sources, it is not a recommended way of training neural networks because it is better to change it manually. [Kar19]

4.6 Embedding and using neural language models in mobile applications

Embedding and using a neural language model in an iOS application is a complex task, as there is no detailed official documentation. First, the model must be converted to the .mlmodel format. The model should then be imported into the iOS application project. To make a prediction with the model in the iOS application, it is necessary to use the CoreLM framework. The text generation process can be done by following:

- Receiving information from the user.
- Mapping the user input to a tensor.
- Forward propagation with a given tensor as an input.
- As a result, the vector of probabilities will be created, where each probability corresponds to a character.
- Drawing a sample from a produced probability distribution, resulting in a predicted character.
- Concatenating the predicted character with the input.
- Dropping the first character in a resulted string.
- Feeding the newly created string to the network and repeating the process.

4.7 User interface

The user interface of the mobile application has to be intuitive. To achieve this, it will resemble a messaging app, a paradigm with which most mobile phone owners are familiar.

4.8 Concept conclusion

In summary, the project objective will be achieved by the following:

- Preprocessing datasets.
- Building and training three recurrent language models using the Keras framework with the TensorFlow backend and converting them into the .mlmodel format with the Coremltools Python package.
- Embedding the language models in an iOS application with CoreML framework.
- Building a user interface for the iOS application which resembles a messaging app.

Chapter 5

Implementation

This chapter describes the implementation of a mobile application with an embedded neural language model. First, the datasets will be preprocessed. The language models will then be built, trained, and converted. The models will be embedded into an iOS application, and the last step is to create a user interface for the mobile application.

5.1 Overview

The project is divided into separate folders:

- *data* - contains datasets used in this thesis and the corresponding code for preprocessing.
- *models* - contains language models developed as a part of this thesis and the corresponding code for creating, evaluating and converting models.
- *app* - contains the mobile application project and corresponding resources.

5.2 Preprocessing

The code for preprocessing Shakespeare's works can be found in the data folder as a Jupyter notebook file. Since character-level models can work with any vocabulary, all that was required was to merge the works into one file and divide it into training, validation, and test sets. The traininig set contains 4,227,036 characters, the validation set contains 528,379 and a test set 528,380 characters, resulting in total of 5,283,795 characters. Preprocessing is done by using the Python programming language and its standard packages.

```
def preprocess(combined_file_name, path_to_raw_folder,
    path_to_finish_folder):
    combine_files(combined_file_name, path_to_raw_folder)
    text = read_text_file(combined_file_name)
    training, validation, test = get_training_validation_testing_sets(text)
    write_to_file(path_to_finish_folder + "training.txt", training)
    write_to_file(path_to_finish_folder + "validation.txt", validation)
    write_to_file(path_to_finish_folder + "test.txt", test)
    return training, validation, test
```

The second dataset, Penn Tree Bank, is already available in the preprocessed format. The training set contains 5,101,618 characters, the validation set contains 399,782 and a test set contains 449,945 characters.

5.3 Language models

The code for each text generation model can be found in the *models* folder. This is a Jupyter notebook file named "models.ipynb" that contains the Python code for the models described in the previous chapter. It can be executed locally on any operating system or remotely using Google Colaboratory environment. The language models and its corresponding Python code are partially based on the projects discussed in "Actual models" section of this thesis [4.5.4].

It is necessary to define which model to train (first, second or third), the length of the input, the path to the data, and the path to the embeddings if necessary.

```
MODEL_CHOICE = 1
MAXLEN = 40
TRAINING_DATA_PATH = "../data/Shakespeare/shakespeare_finish/training.txt"
VALIDATION_DATA_PATH =
    "../data/Shakespeare/shakespeare_finish/validation.txt"
TEST_DATA_PATH = "../data/Shakespeare/shakespeare_finish/test.txt"
EMBEDDINGS = "../data/Embeddings/glove.840B.300d-char.txt"
```

Dictionaries with characters' keys are then created. This creates a mapping between characters and the numerical values which represent these characters. The function *lower* was applied to the text in order to minimize the size of the vocabulary.

```
with io.open(TRAINING_DATA_PATH, encoding='utf-8') as f:
    training_text = f.read().lower()

chars = sorted(list(set(training_text)))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

Function *prepare_data* is responsible for splitting the data into inputs x and true labels y.

```
def prepare_data(path, embeddings_path = EMBEDDINGS):
    ...
    sentences = []
    next_chars = []
    for i in range(0, len(text) - MAXLEN, STEP):
        sentences.append(text[i: i + MAXLEN])
        next_chars.append(text[i + MAXLEN])

    if (MODEL_CHOICE == 1) or (MODEL_CHOICE == 2):
        x = np.zeros((len(sentences), MAXLEN, len(chars)), dtype=np.bool)
        y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
        for i, sentence in enumerate(sentences):
            for t, char in enumerate(sentence):
                x[i, t, char_indices[char]] = 1
            y[i, char_indices[next_chars[i]]] = 1
  ...
```

Function *build_and_compile_model* builds a sequential Keras model based on the user's choice. The function also adds the Adam optimizer into the model and defines a loss function, which is categorical crossentropy (negative log likelihood).

```
def build_and_compile_model():
    model = Sequential()
    if MODEL_CHOICE == 1:
        model.add(LSTM(128, input_shape=(MAXLEN, len(chars))))
        model.add(Dense(len(chars), activation='softmax'))
    elif MODEL_CHOICE == 2:
     ...
    optimizer = Adam(lr=0.0002, beta_1=0.9, beta_2=0.999, epsilon=None,
        decay=0.0)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer,
        metrics=['accuracy', 'crossentropy'])
    return model
```

Function *generate_text* produces a text sample. It takes a sentence from the text data that will be used as an input to the model, and creates a tensor the model can accept.

A prediction is made by calling the corresponding function on the model after passing the tensor as an argument. The model returns the probability distribution over the entire vocabulary, draws a sample from this distribution and prints the output.

```
def generate_text(model, text, number_of_characters):
  ...
        if (MODEL_CHOICE == 1) or (MODEL_CHOICE == 2):
            for i in range(number_of_characters):
                x_pred = np.zeros((1, MAXLEN, len(chars)))
                for t, char in enumerate(sentence):
                    x_pred[0, t, char_indices[char]] = 1.

                preds = model.predict(x_pred, verbose=0)[0]

                next_index = sample(preds, diversity)
                next_char = indices_char[next_index]
                generated += next_char
                sentence = sentence[1:] + next_char
                ...
```

Function *bits_per_character* calculates the *bits per character* metric on a given text corpus.

```
def bits_per_character(string, model):
    ...
    if (MODEL_CHOICE == 1) or (MODEL_CHOICE == 2):
        x = np.zeros((len(sentences), MAXLEN, len(chars)))
        for i, sentence in enumerate(sentences):
            for t, char in enumerate(sentence):
                x[i, t, char_indices[char]] = 1.0
        preds = model.predict(x)
        for i, (sentence, next_char) in enumerate(zip(sentences,
            next_chars)):
            next_ind = char_indices[next_char]
            p.append(preds[i, next_ind])
    return np.mean(-np.log2(p))
```

To build and train the models, the following code needs to be executed. We use functions *prepare_data* to create a training set, *build_and_compile_model* to create a model and a standard Keras function *fit* for training.

```
text_train, x_train, y_train, embedding_matrix =
    prepare_data(TRAINING_DATA_PATH)
model = build_and_compile_model()
...
model.fit(x = x_train, y = y_train,
          batch_size=128,
          epochs=100,
          callbacks=[print_callback])
```

To evaluate a model on the test set, we need to load a model and then run *bits_per_character* function, passing test data, and the model as arguments.

```
MODEL_CHOICE = 1
model = load_model("keras_models/1_model_shakespeare.h5")
print(bits_per_character(test_text, model))
```

5.4 Conversion to CoreML format

The code for model conversions from regular Keras format to CoreML format can be found in the *models* folder as well. It is a Jupyter notebook file named “conversion.ipynb”. It can only be executed on the macOS operating system due to the requirements of the Coremltools package.

```
model = load_model("keras_models/1_model_shakespeare.h5")
coreml_model = coremltools.converters.keras.convert(model)
coreml_model.save("converted_core_ml_models/my_model_shakespeare.mlmodel")
```

5.5 Embedding a model in a mobile application

Folder *app* contains the mobile application project and corresponding resources. The code responsible for generating text samples can be found in a *Presenter* class. The application is written in the Swift programming language.

Function *generate* takes the input from the user and invokes a function *startProducing* to generate a character in a loop.

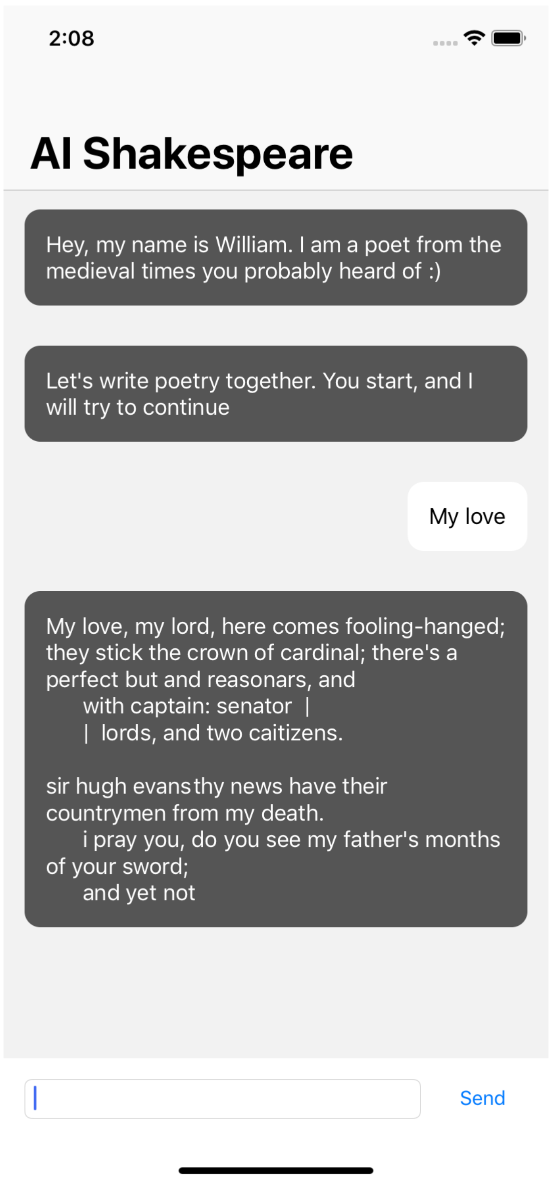
```
func generate(prompt: String, onSuccess: (String) -> (), onError: (Error)
    -> ()) {
    ...
    let sentence = preprocessTextInput(prompt)
    startProducing(sentence: sentence, onSuccess: onSuccess, onError:
        onError)}
```

Function *startProducing* makes a prediction by calling the corresponding function *produce*, passing the input tensor as an argument. The model returns the probability distribution over the entire vocabulary, and a sample is drawn from this distribution. The function returns the final text to the caller in order for display.

```
func startProducing(sentence: String, onSuccess: (String) -> (), onError:
    (Error) -> ()) {
      ...
    let model_Input = keras_modelInput(input1: input_data0)
            ...
            for (index, char) in sentence.enumerated() {
                input1[[NSNumber(value: index),0, 0]] = NSNumber(value:
                    Constants.chars_index_dictionary[char]!)
            }
            ...
            let (char, prediction) = produce(model_Input)
            model_Input.lstm_1_c_in = prediction.lstm_1_c_out
            model_Input.lstm_1_h_in = prediction.lstm_1_h_out
            globalSentence = globalSentence + char
            sentence = (sentence.dropFirst() + char)
            if (globalSentence.count == numberOfCharatersToProduce) {
                onSuccess(globalSentence) }}
```

5.6 User interface

The user interface, which reflects common messaging apps, is built using the standard iOS tool AutoLayout.



Hey, my name is William. I am a poet from the medieval times you probably heard of :)

Let's write poetry together. You start, and I will try to continue

My love

My love, my lord, here comes fooling-hanged; they stick the crown of cardinal; there's a perfect but and reasonars, and
    with captain: senator |
    | lords, and two caitizens.

sir hugh evansthy news have their countrymen from my death.
    i pray you, do you see my father's months of your sword;
    and yet not

Figure 5.1: User interface

Chapter 6

Evaluation

In this chapter, the performance of the models will be presented and evaluated, both on a *bits per character* metric and the on-device inference time.

6.1 Generated text

6.1.1 Works of W. Shakespeare

The models were trained for 50 hours until the loss value stopped improving. During the training, the learning rate was decreased manually.

Given the string “My love” the models produce the following texts:

- First model: "My love is the lady of the will in the stranger of the roman to the country for the direward and men that i have for the more hath so so like a grace of the store that i have so some bears and the store, where is a man to him the words and with beauty of this will be a charge and the fair bloody be an of"
- Second model: "My love, the lady hath made a man of his seat, that i have slaved his daughter in the case, and a second heart to please your mind to be the streets, which is a state, to speak and sight on him."
- Third model: "My love he shall be patient, and now alter’d in england’s thing that thee are like lucentio, if you see him to do them shall be such a name; and the latter is a woman’s hand i may be as much left in hard blood in the court of my life."

As we can see the models can, to some degree, capture Shakespeare’s writing style and produce, in some cases, relevant text samples. Unfortunately, in many cases, the texts are not comprehensible, especially after 40th character. The models do not capture long-range dependencies.

6.1.2 Penn TreeBank

The models were trained for 25 hours until the loss value stopped improving. During the training, the learning rate was decreased manually. Given the string “For weeks the market” the models produce the following texts:

- First model: "For weeks the market research products are problems of company in the more than the manager the company 's a $< unk >$ mat the dollar reselted the proposal deal of the other company and $< unk >$ used to $< unk >$ the proposal is sell for the campuer of the $< unk >$ because of the first had a $< unk >$ the reserves the $< unk >$ $< unk >$"
- Second model: "For weeks the market interest rates the company said it expects to be selling the trading company was n't $< unk >$ and $< unk >$ a computer with a statement the financial institutions were the company said the first department stock at least the company 's $< unk >$ law and the soviet entire the new york stock exchange an"
- Third model: "For weeks the market new trading division of the $< unk >$ concern the treasury department said the dividend of the $< unk >$ of chief executive officer of the decline is $< unk >$ $< unk >$ of the court and the $< unk >$ maintenance communications in the past two years mr. letin which has expected to include the board 's $< unk >$ pr"

Since the training text corpus is not structured (random samples), the models' generated text in many cases is incoherent. However, the models perform much better on this dataset as we will see in the next section.

6.2 Bits per character and inference evaluation

The metric used to evaluate the models is *bits per character* (BPC). The models are evaluated on test sets. The inference measurement for 300 character generation is done by using iPhone 6.

6.2.1 Works of W. Shakespeare

The BPC metric shows us that the models are far from state-of-the art, which is usually around 1.1 BPC. [Rud19]. However, such a high value is influenced not only by the small sizes of the models but also by the training data. Poetry generally has more complicated structure than the news articles. The inference time is within range of what a user can expect from such an application, except for the second model. Overall the best model is the model that used character embeddings.

Model	**BPC**	**Inference**	**Params**
First model (LSTM-128)	2.302	2.53s	97,457
Second model (LSTM-512)	2.44	13.25s	3,275,313
Third model (LSTM-256 with embeddings)	2.26	2.22s	875,470

Table 6.1: Comparison of *bits per character* metric and inference time on different models that were trained on the works of W. Shakespeare

6.2.2 Penn TreeBank

As we can see, the models perform much better on the Penn TreeBank dataset than on the works of W. Shakespeare. Penn TreeBank is a collection of Wall Street Journal samples. The syntax and semantic rules of the dataset are much simpler than Shakespeare's poetry. The models are much closer to the state-of-the-art, especially the third model, though the difference is still significant. The inference time is similar to the one reported on the works on W. Shakespeare.

Model	**BPC**	**Inference**	**Params**
[MMS17, p. 5] FS-LSTM-4	1.193	no data	6.5M
[MMS17, p. 5] FS-LSTM-2	1.190	no data	7.2M
First model (LSTM-128)	1.902	2.67s	97,457
Second model (LSTM-512)	1.510	11.83s	3,275,313
Third model (LSTM-256 with embeddings)	1.457	2.15s	875,470

Table 6.2: Comparison of *bits per character* metric and inference time on different models that were trained on the Penn TreeBank dataset. This table include two state-of-the-art models for the dataset for better comparison

6.3 Summing up

As we can see, while the models generally capture the syntax of the language and the writing style, it is still in many cases not comprehensible. The *bits per character* metric shows us that the models are far from state-of-the-art.

The reason for this is that the models are very small, a necessity when running on mobile devices. The results are nevertheless very promising, given the fact that the mobile models are compared to large state-of-the-art models with many more parameters.

In addition, inference completes in a reasonable amount of time, acceptable for a production environment.

Character embeddings significantly improve the scores of a model. The best model in terms of BPC and inference time on both datasets is the model which used character embeddings that are derived from GloVe word vectors.

Chapter 7

Conclusion

This chapter summarizes the work done in this thesis.

7.1 Summary

In this work, we tried to answer the question of whether it is possible to generate text on mobile devices. For this purpose, the leading mobile platforms, tools, and frameworks were analyzed. Several text generation models suitable for mobile devices were built, trained, and evaluated on various text corpora. We used character-level models because they have fewer parameters and therefore are smaller than word-level models, which is essential for mobile devices' inference time. We also placed a limit on the number of parameters the model can have for the same reason. We chose to use recurrent neural networks because of their popularity for language modeling, and also because of the technical possibility of deployment to mobile devices. In order to test a variety of models, each model had a key feature: a small network, a larger network, and a network with character embeddings.

We analyzed both Android and iOS platforms and decided to deploy the models to iOS mobile devices because of the better support for models conversion to mobile format.

Eventually, a mobile application with an embedded neural language model was developed. It showed us that it is not only possible to generate text samples on mobile devices, but also that these samples can resemble the training data, thus giving the generated text the original author's style.

We can conclude that the appropriate number of parameters of the model, suited for mobile deployment, is between 50,000 - 1,000,000. We made this conclusion since only models with 97,457 and 875,470 parameters give appropriate inference time.

The model that uses character embeddings achieves the best score, both in language modeling metric as well as inference time, thus proving the point that embeddings significantly improve models' performance.

We found out that training datasets highly influence the results of models. The same models have lower scores when training data is medieval poetry, and higher scores when it is news articles.

The approach of this thesis applies to any sequence prediction task that can be done on mobile, such as machine translation and text summarization because they can be performed in a very similar fashion.

The findings of this thesis may be useful to any organization that wants to offload the work to mobile, therefore improving security, user's privacy, and latency.

The possible applications are offline working mobile text generators (writer helpers), conversational bots, FAQ-bots and support bots, translators, and text summarizers.

7.2 Critical review

Unfortunately, none of the models achieves state-of-the-art results or captures long-range dependencies. Because we execute the models on mobile devices, the models have to be small and have few parameters. Therefore, they are not deep and powerful enough to achieve state-of-the-art results.

One of the solutions could be to conduct additional research on small yet powerful models that could be converted to mobile format. Other mobile machine learning frameworks could be explored as well, for example, the MACE framework. [19l] Different kind of regularizations could be tried too.

Another approach to improve the models' scores might be to try word-level models with short vocabularies, e.g., narrow the scope of text generation. Instead of trying to capture whole Shakespeare's style, we could use small text dataset for some particular task, like short conversations. Due to the time restrictions, this lies out of the scope of this work.

The second problem of the project is that training data was not fully preprocessed to suite the user interface of a mobile application. The Shakespeare's text data contains large chunks of empty spaces and a lot of blank lines, which influenced the generated output. As a result, the screen of the mobile device was not large enough to fit the text samples.

Additionally, the models' output does not contain any uppercase letters. The training data was lowercased in order to minimize the size of the vocabulary. However, this made the generated text samples hard to read. Therefore, it is probably better to avoid such modifying of the training data.

7.3 Future work

As was mentioned, possible future work involves finding and implementing other types of models. For example, quasi-recurrent neural networks have very promising results. [TL18]

Consideration could also be given to adapting attention-based models, such as Transformers, for use in mobile applications. Most of them have a considerable number of parameters, so it is necessary to consider quantization of the large models to smaller ones while maintaining their efficiency so that they can be deployed to mobile devices while retaining acceptable performance.

List of Figures

List of Tables

Appendix A

A.1 Contents

A USB flash drive is submitted together with this thesis. It contains the source code, online sources and the thesis as a PDF file.

References

[15] *Introducing the world's first neural network keyboard.* 2015. URL: https://blog.swiftkey.com/neural-networks-a-meaningful-leap-for-mobile-typing (visited on 08/08/2019).

[18a] *GitHub issue: How to use LSTM layer in tfLite?* 2018. URL: https://github.com/tensorflow/tensorflow/issues/20881 (visited on 08/08/2019).

[18b] *GitHub issue: Unable to convert LSTM model to .tflite model.* 2018. URL: https://github.com/tensorflow/tensorflow/issues/15805 (visited on 08/08/2019).

[19a] *About Swift.* 2019. URL: https://swift.org/about/ (visited on 08/08/2019).

[19b] *Android Machine Learning Documentation.* 2019. URL: https://developer.android.com/ml (visited on 08/08/2019).

[19c] *Apple Machine Learning Documentation.* 2019. URL: https://developer.apple.com/machine-learning (visited on 08/08/2019).

[19d] *CoreML Documentation.* 2019. URL: https://developer.apple.com/documentation/coreml (visited on 08/08/2019).

[19e] *Coremltools Documentation.* 2019. URL: https://apple.github.io/coremltools/ (visited on 08/08/2019).

[19f] *CreateML Documentation.* 2019. URL: https://developer.apple.com/documentation/createml (visited on 08/08/2019).

[19g] *Google Colaboratory F.A.Q.* 2019. URL: https://research.google.com/colaboratory/faq.html (visited on 08/08/2019).

[19h] *Information about the app Continuous.* 2019. URL: http://continuous.codes (visited on 08/08/2019).

[19i] *Keras Documentation.* 2019. URL: https://keras.io (visited on 08/08/2019).

[19j] *Keras LSTM text generation example.* 2019. URL: https://github.com/keras-team/keras/blob/master/examples/lstm_text_generation.py (visited on 08/08/2019).

[19k] *Kotlin overview.* 2019. URL: https://developer.android.com/kotlin/overview (visited on 08/08/2019).

[19l] *MACE framework Source Code.* 2019. URL: https://github.com/XiaoMi/mace (visited on 08/08/2019).

[19m] *Making Predictions with a Sequence of Inputs.* 2019. URL: https://developer.apple.com/documentation/coreml/core_ml_api/making_predictions_with_a_sequence_of_inputs (visited on 08/08/2019).

[19n] *Python F.A.Q.* 2019. URL: https://docs.python.org/3/faq/general.html#what-is-python (visited on 08/08/2019).

[19o] *SwiftKey app information.* 2019. URL: https://www.microsoft.com/en-us/swiftkey/about-us (visited on 08/08/2019).

[19p] *TensorFlow Documentation.* 2019. URL: https://www.tensorflow.org (visited on 08/08/2019).

[19q] *TensorFlow Keras Module Documentation.* 2019. URL: https://www.tensorflow.org/guide/keras (visited on 08/08/2019).

[19r] *TensorFlow Lite Converter Documentation.* 2019. URL: https://www.tensorflow.org/lite/convert (visited on 08/08/2019).

[19s] *TensorFlow Lite Documentation.* 2019. URL: https://www.tensorflow.org/lite (visited on 08/08/2019).

[BZ19] Terra Blevins and Luke Zettlemoyer. *Better Character Language Modeling Through Morphology.* 2019. arXiv: 1906.01037. URL: https://arxiv.org/abs/1906.01037.

[Cao17] Qingqing Cao. *MobiRnn Source Code.* 2017. URL: https://github.com/csarron/MobiRnn (visited on 08/08/2019).

[Elm90] Jeffrey L. Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211. URL: https://crl.ucsd.edu/~elman/Papers/fsit.pdf (visited on 08/08/2019).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* https://www.deeplearningbook.org/contents/rnn.html. MIT Press, 2016.

[Gör17] Martin Görner. *tensorflow-rnn-shakespeare text files.* 2017. URL: https://github.com/martin-gorner/tensorflow-rnn-shakespeare/tree/master/shakespeare (visited on 08/08/2019).

[Gra13] Alex Graves. *Generating Sequences With Recurrent Neural Networks.* 2013. arXiv: 1308.0850. URL: https://arxiv.org/abs/1308.0850.

[HB19] Michael Hahn and Marco Baroni. *Tabula nearly rasa: Probing the Linguistic Knowledge of Character-Level Neural Language Models Trained on Unsegmented Text.* 2019. arXiv: 1906.07285. URL: https://arxiv.org/abs/1906.07285.

[HS16] Kyuyeon Hwang and Wonyong Sung. *Character-Level Language Modeling with Hierarchical Recurrent Neural Networks.* 2016. arXiv: 1609.03777. URL: https://arxiv.org/abs/1609.03777.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[IS15] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: 1502.03167. URL: https://arxiv.org/abs/1502.03167.

[JM18] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (3rd Edition).* 2018. URL: https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf (visited on 08/08/2019).

[Joz16] Rafal Jozefowicz et al. *Exploring the Limits of Language Modeling.* 2016. arXiv: 1602.02410. URL: https://arxiv.org/abs/1602.02410.

[Kar15] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks.* 2015. URL: https://karpathy.github.io/2015/05/21/rnn-effectiveness (visited on 08/08/2019).

[Kar19] Andrej Karpathy. *A Recipe for Training Neural Networks.* 2019. URL: https://karpathy.github.io/2019/04/25/recipe/ (visited on 08/08/2019).

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[Kim15] Yoon Kim et al. *Character-Aware Neural Language Models.* 2015. arXiv: 1508.06615. URL: https://arxiv.org/abs/1508.06615.

[Kru18] Frank A. Krueger. *C# Code Prediction with a Neural Network.* 2018. URL: https://praeclarum.org/2018/07/20/code-prediction-with-a-neural-network.html (visited on 08/08/2019).

[Mik10] Tomas Mikolov. *Penn TreeBank Dataset.* 2010. URL: https://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz (visited on 08/08/2019).

[Mik12] Tomas Mikolov et al. *Subword language modeling with neural networks.* 2012. eprint: preprint. URL: https://www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf.

[MKS18] S. Merity, N. Shirish Keskar, and Richard Socher. *An Analysis of Neural Language Modeling at Multiple Scales.* 2018. arXiv: 1803.08240. URL: https://arxiv.org/abs/1803.08240.

[MMS17] Asier Mujika, Florian Meier, and Angelika Steger. *Fast-Slow Recurrent Neural Networks.* 2017. arXiv: 1705.08639. URL: https://arxiv.org/abs/1705.08639.

[MSM93] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a large annotated corpus of english: The penn treebank." In: *COMPUTATIONAL LINGUISTICS* (1993), pp. 313–330. URL: https://repository.upenn.edu/cgi/viewcontent.cgi?article=1246&context=cis_reports.

[NH10] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines". In: *ICMLS* (2010). URL: https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP).* 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[Qiu16] Liang Qiu. *GRU_Android Source Code.* 2016. URL: https://github.com/Liang-Qiu/GRU_Android (visited on 08/08/2019).

[Rad19] A. Radford et al. *Language Models are Unsupervised Multitask Learners.* 2019. URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (visited on 08/08/2019).

[Rud19] Sebastian Ruder et al. *NLP-progress.* 2019. URL: https://github.com/sebastianruder/NLP-progress/blob/master/english/language_modeling.md (visited on 08/08/2019).

[SG15] Cicero Nogueira dos Santos and Victor Guimarães. *Boosting Named Entity Recognition with Neural Character Embeddings.* 2015. arXiv: 1505.05008. URL: https://arxiv.org/abs/1505.05008.

[SHB15] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2015. arXiv: `1508.07909`. URL: `https://arxiv.org/abs/1508.07909`.

[Sri14] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *Journal of Machine Learning Research* 15(1) (2014), pp. 1929–1958. URL: `www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf`.

[TL18] Raphael Tang and Jimmy Lin. *Adaptive Pruning of Neural Language Models for Mobile Devices*. 2018. arXiv: `1809.10282`. URL: `https://arxiv.org/abs/1809.10282`.

[TO17] Corentin Tallec and Yann Ollivier. *Unbiasing Truncated Backpropagation Through Time*. 2017. arXiv: `1705.08209`. URL: `https://arxiv.org/abs/1705.08209`.

[Woo] M. Woolf. *char-embeddings Source Code*. URL: `https://github.com/minimaxir/char-embeddings` (visited on 08/08/2019).