Norwegian University of
Science and Technology

Wake Word Detection Using Recurrent Neural Networks

Bjørnar Hunshamar

Master of Science in Communication Technology
Submission date: June 2018
Supervisor: Torbjørn Svendsen, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science (MSc) in Communication Technology at the Norwegian University of Science and Technology. The project was carried out during the spring of 2018 in cooperation with Cisco Systems Norway AS and supervised by professor Torbjørn Karl Svendsen. The project was done independently, but weekly meetings with supervisors from Cisco were held to evaluate progress and discuss further work.

I would like to thank professor Torbjørn Karl Svendsen, as well as Svein Gunnar Pettersen and Haakon Sandsmark from Cisco Systems AS, for their supervision, guidance, and feedback.

Also thanks to Cisco Systems AS for providing the data set used to train the models presented in this thesis.

Trondheim, June 2018 Bjørnar Hunshamar

Abstract

Wake word detection is the process of continuously listening for some specific keyword or phrase which, when uttered, wakes up a system and initiates communication between the system and a user. This is a necessity for interaction with virtual assistants such as Google Now, Apple's Siri, Microsoft's Cortana, and Amazon's Alexa.

The goal of this project was to implement and evaluate a wake word detection algorithm based on deep learning. Two systems were implemented using recurrent neural networks which were trained as binary classifiers designed to return a confidence score indicating whether a wake word was uttered or not within a speech frame. The first system was based on a commonly used approach in which features are extracted from a sliding window of fixed length. The second system was more experimental, and exploited some of the capabilities of recurrent neural networks in order to eliminate the need for overlapping and fixed-length windows.

Both systems used GRU networks with 65,154 trainable parameters, and thus had a similar memory footprint. However, the experimental system cut the computation cost by nearly 80 % compared to the sliding window based system, while also reducing the number of false detections. The results were highly encouraging, though there is still room for improvement. The first step would be using a larger and more diverse data set, and using a higher ratio of negatives to positives. Added layers, tweaking of parameters, regularization techniques or using a different type of cell in the network might also have a positive effect.

Sammendrag

Vekkeorddeteksjon innebærer å lytte etter et bestemt nøkkelord eller -ytring i en kontinuerlig talestrøm. Ved deteksjon skal systemet vekkes for videre kommunikasjon med brukeren. Denne teknologien er en nødvendighet for interaksjon med virtuelle assistenter som Google Now, Apples Siri, Microsofts Cortana og Amazons Alexa.

Målet med dette prosjektet var å implementere og evaluere en algoritme for vekkeorddeteksjon basert på dyp læring. To systemer ble implementert ved bruk av rekurrente nevrale nettverk, som ble trent som binære klassifiserere designet til å returnere en sannsynlighetsverdi for at et vekkeord har blitt ytret. Det første systemet brukte en velkjent tilnærming hvor egenskapsvektorer blir beregnet basert på et glidende vindu. Det andre systemet var mer eksperimentelt, og brukte noen av fordelene med rekurrente nevrale nettverk til å eliminere behovet for overlappende tidsvinduer av fast lengde.

Begge systemene brukte GRU-nettverk med 65 154 trenbare parametre, og hadde dermed ganske lik bruk av minne. Det eksperimentelle systemet reduserte derimot prosessorbruken med nærmere 80 %, samtidig som at færre falske deteksjoner ble registrert. Resultatene var meget oppmuntrende, men det er fremdeles rom for forbedring. Første steg vil være å bruke et større og mer variert datasett, og å ha en større andel av negativer. I tillegg bør det også være mulig å forbedre modellen ved å legge til lag, justere parametere, benytte seg av regulariseringsteknikker og teste ut andre typer celler i nettverket.

Table of Contents

Preface i

Abstract ii

Sammendrag iii

Table of Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Project Description 2
- 1.3 Thesis Outline 2

2 Theory 4
- 2.1 Automatic Speech Recognition 4
  - 2.1.1 Overview 4
  - 2.1.2 MFCC Feature Extraction 5
- 2.2 Introduction to Artificial Neural Networks 7
  - 2.2.1 Artificial Neurons 8
  - 2.2.2 Multilayer Perceptrons 9
  - 2.2.3 Training a Neural Network 10
  - 2.2.4 Performance Measures 11
- 2.3 Recurrent Neural Networks 13
  - 2.3.1 Basic Architecture 13
  - 2.3.2 GRU Cell 15

3 Tools and Methods 18
- 3.1 Software Tools and Data 18
  - 3.1.1 TensorFlow 18
  - 3.1.2 Other Modules and Libraries 19

3.1.3 Data Set 19
3.2 Sliding Window Approach 20
3.2.1 Pre-processing 20
3.2.2 RNN Model Architecture 22
3.2.3 System Implementation 23
3.3 Non-overlapping Window Approach 24
3.3.1 Pre-processing 25
3.3.2 RNN Model Architecture 26
3.3.3 System Implementation 27

4 Results and Discussion **29**
4.1 Sliding Window Approach 30
4.1.1 RNN Model Performance 30
4.1.2 System Performance 31
4.2 Non-overlapping Window Approach 33
4.2.1 RNN Model Performance 33
4.2.2 System Performance 34
4.3 Comparison 36

5 Conclusion **38**
5.1 Summary 38
5.2 Future Work 39

References **40**

List of Figures

2.1 Training a classifier for speech recognition, and using it to predict the label of some spoken utterance. 5
2.2 Overview of MFCC computation. 7
2.3 An artificial neuron with three inputs. 8
2.4 Sigmoid, tanh and ReLU outputs plotted for values between -6 and 6. 9
2.5 A simple multilayer perceptron with three input neurons, four hidden neurons, and one output neuron. 10
2.6 An RNN cell unrolled through three time steps. 14
2.7 An unrolled sequence-classifying RNN. 15
2.8 The structure of a GRU cell[1]. 16

3.1 Original negative speech signal, randomly selected slice, and loudest frame. 21
3.2 A positive utterance before and after pre-processing. 22
3.3 GRU model unrolled through 149 time steps. 23
3.4 Three consecutive windows of a speech signal. The third frame contains the whole utterance. 24
3.5 Speech signal before and after removing trailing silence. 25
3.6 Overview of the model used for the system based on non-overlapping windows. 27
3.7 Continuous wake word detection overview. 28

4.1 Varying precision and recall for the sliding window based model. 31
4.2 Plotted predictions from three files using the sliding window approach. 32
4.3 Varying precision and recall for the non-overlapping window based model. 34
4.4 Plotted predictions from three files using the non-overlapping window approach. 35

List of Tables

2.1 Example confusion matrix of a faulty model. 12

4.1 Confusion matrix for the sliding window based model. 30
4.2 Precision and recall values for the sliding window based model. . 31
4.3 Confusion matrix for the non-overlapping window based model. . 33
4.4 Precision and recall values for the non-overlapping window based model. 34
4.5 Precision and recall comparison for the two models. 36
4.6 Comparison of the two implemented systems. 37

Chapter 1

Introduction

1.1 Motivation

Voice interaction with technology is becoming increasingly commonplace due to the rapid development of smartphones and tablets, as well as artificial intelligence. This allows communication with a computer without using a keyboard.

In order for a hands-free system to initiate voice input, it needs to continuously listen for some specific keyword. Virtual assistants such as Google Now, Apple's Siri, Microsoft's Cortana, and Amazon's Alexa can all be addressed by saying a wake word, such as "OK Google", "Hey Siri", or "Alexa", followed by some voice command. The process of listening for this wake word is called wake word detection.

A wake word detection system needs to have a small memory footprint, low computational cost, and high precision[2]. Traditionally, the Keyword/Filler Hidden Markov Model has been used for wake word detection. However, artificial neural networks have been shown to improve complexity, run-time computation, memory footprint and latency.

Convolutional neural networks have been a popular choice in recent years, as they have been shown to outperform basic multi-layer perceptrons with far fewer parameters[3]. Both of these neural network classes are usually implemented for wake word detection by extracting feature vectors from a sliding window of fixed length, and passing it through the neural network. The produced output is used as a confidence score indicating whether the wake word was uttered inside the window or not. However, the overlapping nature of this approach leads to an excess of computations. Also, it requires a pre-defined length for all training data, i.e. the size of the window.

By instead using a recurrent neural network model, it should be possible to design a system which doesn't require overlapping windows, and which can be trained using variable length training data. This will greatly reduce the computation cost, and ensure that no important data is lost in the process of fitting training data within a window.

1.2 Project Description

In this thesis, two different wake word detection systems based on recurrent neural networks will be designed and implemented. The first system will use the sliding window approach, while the second system will be trained in a way which allows it to continuously listen for wake words without any overlap. Ideally, the latter should perform as well as the former, while significantly reducing the computation cost.

Training a neural network requires pre-recorded and labeled data. The data set used in this thesis has been provided by Cisco Systems AS. The neural networks will be implemented using TensorFlow in Python language. The two systems will be evaluated and compared using various performance measures, as well as a simulation of how they would perform in a real-time application.

1.3 Thesis Outline

The rest of the thesis is structured as follows.

Chapter 2 provides the basic theory needed to understand and solve the project task. The first section gives an introduction to the field of automatic speech recognition, including a description of how mel-frequency cepstral coefficients can be extracted from a speech signal. The second section gives a general introduction to artificial neural networks, including how they can be designed, how they can be trained, and how they can be evaluated using various performance measures. In the third section, recurrent neural networks are explained, with an emphasis on the GRU cell, which is used in both systems implemented in this thesis.

Chapter 3 describes the tools and methods used to implement the systems. The first section presents the tools used for implementing the systems, as well as the data set used to train the neural network models. The second section provides a thorough description of the implementation of the sliding window approach. The third section describes the implementation of the non-overlapping window approach.

In chapter 4, the two systems are evaluated and compared to each other. Both systems are also applied to longer speech files to simulate how they would work

in a real-time application.

The conclusion is given in chapter 5, along with suggestions for future work.

Chapter 2

Theory

In this chapter, there will be a presentation of the theoretical knowledge needed to understand and solve the project task. It is assumed that the reader has basic knowledge of signal processing and classification.

2.1 Automatic Speech Recognition

The goal of automatic speech recognition is to develop an automated process in which a computer is able to recognize speech or to translate it into text. In the case of wake word detection, this process involves recognizing a chosen word or phrase in a continuous audio stream.

2.1.1 Overview

Automatic speech recognition generally involves training a classifier to recognize some unit of speech. In order to do this accurately, one unit must not be easily confused with other units. The most common unit of speech for speech recognition is either words or phonemes. Phonemes are considered the most basic unit of speech, and they comprise a relatively small set. English as it is spoken in the United States contains 40 phonemes in total[4].

For a general classifier, phonemes are a reasonable choice as speech unit, as they can infer other units such as words or sentences. However, if the number of classes one wants to recognize is small, words may be a better choice, as they generally provide higher accuracy[4]. Wake word detection can be viewed as a binary classification process, where the wake word is classified as a positive and all other utterances are classified as negatives.

To train a classifier, training examples for every class is required. Ideally, training data should be collected from a database which offers a good variety of speakers with regards to age, sex, nationality, and so on. It also needs to contain enough data so that the parameters of the model can be accurately estimated. Usually, a set of features extracted from the waveform is used to represent the unit. The process of feature extraction is described in greater detail in the next subsection.

There are multiple possible choices of classifiers in speech recognition. Neural network classification is described in detail in the second section of this chapter. Other types of classification are beyond the scope of this thesis.

In summary, in order to train a classifier for speech recognition, one needs to collect transcribed data from a speech database, extract features from the waveforms and assign labels indicating the classes of the units, and use this information to estimate the parameters of a model. Once this is done successfully, the model should be able to predict the classes of unlabeled data. The two stages of this process, called training and inference respectively, are illustrated in figure 2.1.

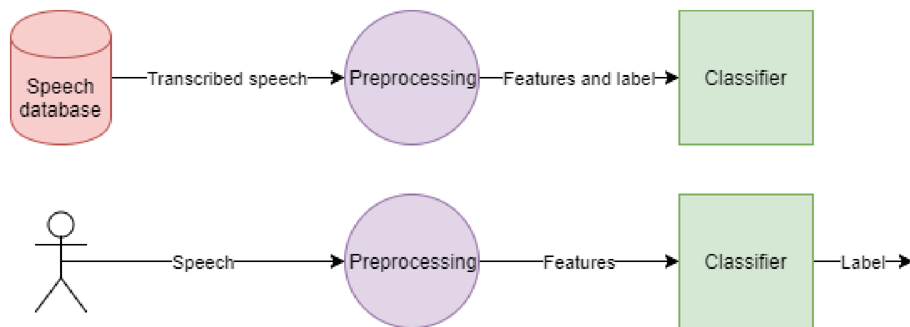


Figure 2.1: Training a classifier for speech recognition, and using it to predict the label of some spoken utterance.

2.1.2 MFCC Feature Extraction

Usually, it is preferable to train the classifier using a compact representation of the speech signal, as opposed to using the waveform itself. This representation, or set of features, needs to be as different as possible for different classes, and as similar as possible within the same class. Features in speech recognition are usually based on the spectrum of the signal, as many features such as formants are better characterized in the frequency domain with a low-dimension feature vector[4].

The mel-frequency cepstral coefficients (MFCC) is a speech representation motivated by the behavior of the human auditory system which has been success-

fully used in speech recognition. It is defined as the real cepstrum of a windowed short-time signal derived from the FFT of that signal.

First the signal is partitioned into small overlapping windows. The MFCCs of each window are computed as follows[4]. First, the discrete Fourier transform of the frame is computed:

$$X_a[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N},\ 0 \leq k < N \tag{2.1}$$

The powers of the spectrum are mapped into mel-frequency scale using M triangular overlapping filters (m = 1,2,...,M):

$$H_m[k] = \begin{cases} 0, & k < f[m-1] \\ \frac{2(k-f[m-1])}{(f[m+1]-f[m-1])(f[m]-f[m-1])}, & f[m-1] \leq k \leq f[m] \\ \frac{2(f[m+1]-k)}{(f[m+1]-f[m-1])(f[m+1]-f[m])}, & f[m] \leq k \leq f[m+1] \\ 0, & k > f[m+1] \end{cases} \tag{2.2}$$

The mel scale is linear below 1 kHz, and logarithmic above, and one mel is defined as one thousandth of the pitch of a 1 kHz tone. It can be approximated by:

$$B(f) = 1125 \ln(1 + f/700) \tag{2.3}$$

This scale models the sensitivity of the human ear, and improves discriminatory capability between speech segments. Then the log-energy at the output of each filter is computed as follows:

$$S[m] = \ln\left[\sum_{k=0}^{N-1} |X_a[k]|^2 H_m[k]\right],\ 0 \leq m < M \tag{2.4}$$

Finally, compute the discrete cosine transform of the filter outputs to obtain the mel-frequency cepstrum:

$$c[n] = \sum_{m=0}^{M-1} S[m] \cos \pi n(m + 1/2)/M,\ 0 \leq n < M \tag{2.5}$$

M varies for different implementations, but for speech recognition using neural networks, 40 coefficients is a reasonable choice.

The process of computing MFCCs for a signal is illustrated in figure 2.2.

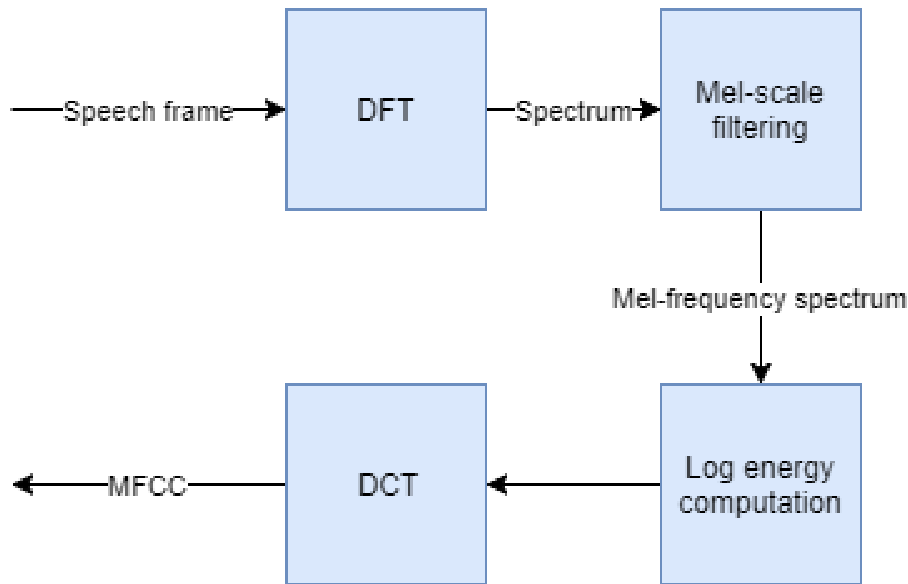


Figure 2.2: Overview of MFCC computation.

The dimensions of an MFCC vector depends on the length of the input signal, and the number of frames it is partitioned into. For example, if each speech unit is of 1-second length, one may choose to divide it into frames of 25 ms length with a 10 ms step size. Then, for each 10 ms, 40 coefficients will be computed based on the following 25 ms of speech. This will result in either a 99x40 matrix or a 100x40 matrix, depending on whether the signal is zero-padded or not respectively.

2.2 Introduction to Artificial Neural Networks

Artificial neural networks (ANN) are computing systems which imitate the biological neurons of the human brain. They are widely used for classification, and have achieved overwhelming success in automatic speech recognition in recent years[5].

The multilayer perceptron is the most basic type of ANN which is simply a mathematical function mapping some set of input values to output values[6]. The function is a combination of simpler functions, each of which provide a new representation of the input.

2.2.1 Artificial Neurons

An artificial neural network is formed by combining artificial neurons. The artificial neuron is simply a mathematical function which computes a weighted sum of its inputs, adds a bias term, and applies some activation function.

$$y = f\Big(\sum_i x_i w_i + b\Big) \tag{2.6}$$

The purpose of the activation function f is to add non-linearity. Without the activation function, combining artificial neurons into more complex structures would serve no purpose, as the result would still be a linear function. By introducing non-linearity to the network, more complex functions can be estimated. In addition, the activation function can limit the contribution of each neuron to the final output.

A single neuron with three inputs is illustrated in figure 2.3.

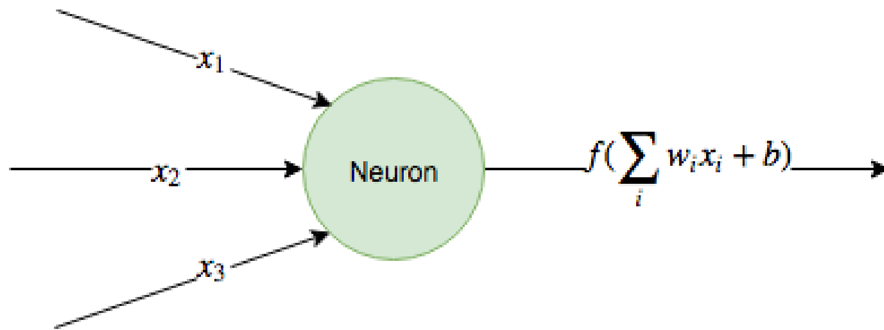


Figure 2.3: An artificial neuron with three inputs.

The most popular activation function is the sigmoid function[5]. The sigmoid function squashes its input to a value between 0 and 1.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.7}$$

The tanh activation function is a version of the sigmoid which is scaled between -1 and 1.

$$\tanh(x) = \frac{2}{1 + e^{-z}} - 1 = 2\sigma(2z) - 1 \tag{2.8}$$

Another popular activation function is the rectified linear unit (ReLU)[5], which simply sets negative values equal to 0.

$$ReLU(z) = max(0, z) \tag{2.9}$$

A comparison of the aforementioned activation functions can be seen in figure 2.4

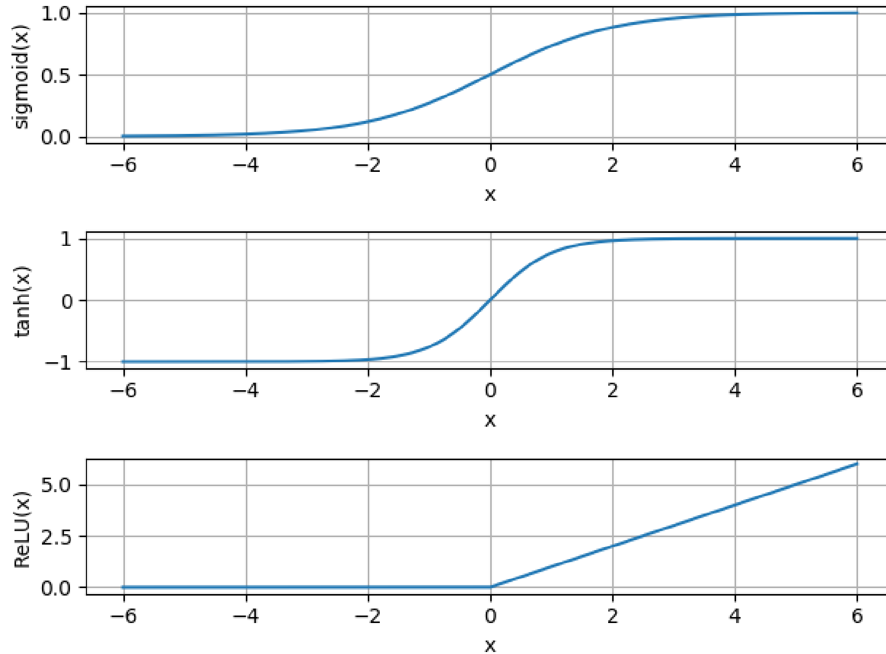


Figure 2.4: Sigmoid, tanh and ReLU outputs plotted for values between -6 and 6.

2.2.2 Multilayer Perceptrons

Multilayer perceptrons are networks of artificial neurons structured into layers. The layers are fully connected, which means that each neuron in one layer is connected to every neuron in the next layer. These models are called feedforward because there are no feedback connections in which outputs of the model are fed back to itself[6].

The inputs to the network are usually referred to as the first layer. They pass through a sequence of hidden layers, each of which perform some transformation before they reach the output layer. For a binary classifier, the output layer may consist of only one neuron, which produces a score indicating which of the two classes the input belongs to. A simplified three-layer binary MLP classifier is illustrated in figure 2.5.

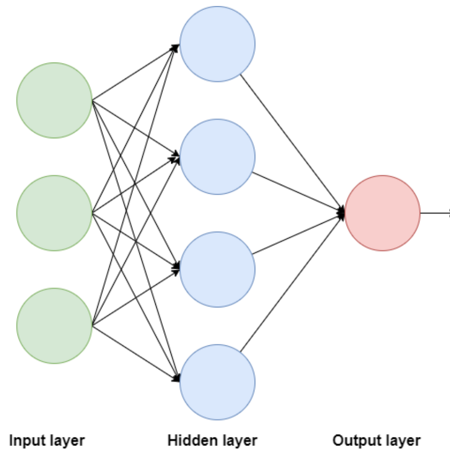


Figure 2.5: A simple multilayer perceptron with three input neurons, four hidden neurons, and one output neuron.

Alternatively, and especially in the case of multi-class classification, the output layer can consist of multiple neurons, each representing a class. Then, instead of applying an activation function on each neuron, the softmax function can be used on the vector of outputs to convert it to a vector of estimated probabilities for each corresponding class [7]. The class probabilities estimated by the softmax function will add up to one. The function is defined as follows:

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp\left(s_k(x)\right)}{\sum_{j=1}^{K} \exp\left(s_j(x)\right)}, \tag{2.10}$$

where K is the number of classes, $s(x)$ is the vector of outputs in the final layer, and $\sigma(s(x))_k$ is the estimated probability that the instance x belongs to class k given the scores of each class for that instance.

Increasing the number of layers and the number of units within a layer increases the complexity of the functions which the network can represent. A multilayer perceptron with multiple hidden layers is called a deep neural network (DNN)[5].

2.2.3 Training a Neural Network

Training a neural network requires a large set of labeled data, which is typically split into three subsets: training data, validation data, and test data. First, the parameters of the model, i.e. the weights and biases, are initialized with random values. Then, the training data is used to fit the model by adjusting the parameters so that the model is able to map its inputs to the right class. In

order to make sure that the model can generalize to new data, the performance is continuously measured by feeding the separate validation data to the network at regular intervals. Finally, the test data is used to give a final evaluation of the performance of the model.

Feeding the training data to the neural network will produce a sequence of confidence score vectors, onto which we can apply the softmax function to get estimated class probabilities. We need to define a cost function in order to quantify how well these probabilities correspond to the target classes. A popular choice is the cross-entropy cost function[7], defined by

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)}) \tag{2.11}$$

where K is the number of classes, $y_k^{(i)}$ is the target class (1 when k is the true class, 0 otherwise), and $\hat{p}_k^{(i)}$ is the class probability vector. The cost is averaged over the total number of training inputs m. θ is the parameter vector, i.e. the weights and biases of the model.

Once the error has been computed, we can use the gradient of the cost function to measure the error contribution from each weight. By considering the biases as an additional neuron in each layer with constant value 1 and separate weights, we can also measure their error contribution. This method is called backpropagation[8], as the error gradient is propagated backward in the network from the output layer to the input layer.

The last step is to minimize the cost function. We can use gradient descent to update the weights in the network according to their error gradients and a scaling factor called the learning rate, which determines the value change in each update. If the learning rate is too low, the training process will be slow. If it is too high, the learning may diverge and never be able to minimize the cost function[9].

The gradient is usually computed based on fixed-size subsets of the training data called batches. A pass through the whole training set is called an epoch. The model is usually trained for multiple epochs, and at the end of each epoch, we measure the average cost of the training data and some performance measure for the validation data. If the training cost decreases, but the validation performance stagnates or decreases, we can stop training, as this indicates that the model has stopped generalizing and started specializing on the training set.

2.2.4 Performance Measures

Once the model has been trained, we need to measure how well it performs using a test subset of the data which has not yet been fed to the network.

The most intuitive performance measure for a classifier is the accuracy of the model, which is simply the ratio of true predictions over total predictions. However, if the data is unbalanced, this will often give misleading results. Consider a binary classifier where one class is regarded as positive and the other as negative. If 90 % of the data belongs to the negative class, predicting this class for all inputs would always give a 90 % accuracy.

A better way to evaluate the performance of such a model is to look at its confusion matrix. The confusion matrix of the example in the previous paragraph, with a total of 1000 examples, would look like this:

	Predicted class 0	**Predicted class 1**
True class 0	900	0
True class 1	100	0

Table 2.1: Example confusion matrix of a faulty model.

This gives more valuable information than a single score. When the true class is 1, a positive prediction and a negative prediction is called a true positive and a false negative respectively. Conversely, when the true class if 0, a positive prediction and a negative prediction is called a true negative and a false positive. An ideal classifier only has true positives and true negatives. In our example, we have 900 true negatives, 100 false negatives, and no positive predictions.

Based on the confusion matrix, we can compute the rate of positive instances correctly classified. Negative instances are not considered. This measure is called the recall of the classifier.

$$Recall = \frac{True\,positives}{True\,positives + False\,negatives} \tag{2.12}$$

The recall is usually used along with another performance measure called precision, which is the accuracy of the positive predictions.

$$Precision = \frac{True\,positives}{True\,positives + False\,positives} \tag{2.13}$$

To summarize, recall is a measure of the model's ability to find all positives within the data, while precision is a measure of the model's ability to only assign the positive label to positive instances. For our example, both the recall and precision will be 0, as there are no true positives. Ideally, both precision and recall should be as close to 1 as possible, and they should be considered as a pair to give valuable information. For example, if we were to classify all the data as positives, the recall would be 1, but the precision would be only 0.1.

Depending on the purpose of the model, one of the measures may be prioritized. In the case of wake word detection, a high recall will ensure that a large number of the wake word utterances will be detected. However, it may lead to a high number of false positives. High precision gives more certainty that the detections correspond to actual wake word utterances. For a system that is continuously listening for a wake word, precision should be the priority, as false detections could wake the system up at unexpected times, and thus be bothersome for the user.

2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks which are specialized for processing sequential data[6]. This is achieved by extending the neural network architecture described in the previous section to include feedback loops, allowing information to persist.

A great advantage of recurrent neural networks is their ability to work on sequences of arbitrary length. Thus, if you're working with speech data, instead of requiring all of the input to have the same dimensions, the network can accept inputs of any length, as long as the dimension of each time step is the same. This simplifies pre-processing, as there is no need to cut or pad sequences, as well as reducing the number of parameters needed, as these are shared across time steps[6].

2.3.1 Basic Architecture

In a basic single-layer RNN, every neuron receives both an input vector $x_{(t)}$ and the output vector from the previous time step $y_{(t-1)}$. This layer of hidden neurons is called a cell. The cell preserves some state across time steps. Since this makes the output at a single time step dependent on all the previous inputs, we say that the cell has memory[9]. A basic RNN is illustrated in figure 2.6.

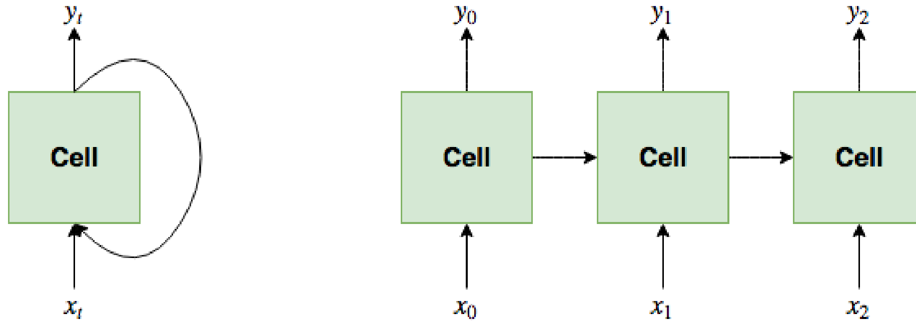


Figure 2.6: An RNN cell unrolled through three time steps.

Each neuron in the hidden layer needs a separate set of weights for the input, and another for the outputs of the previous time step. The outputs of a layer of recurrent neurons for all instances in a batch can be computed as follows[9]:

$$Y_{(t)} = \phi\big(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b\big) \tag{2.14}$$

$Y_{(t)}$ is a two-dimensional matrix containing the outputs of each neuron in the hidden layer for each instance in the batch, and $X_{(t)}$ contains the inputs of each instance. W_x and W_y contains the weights for the inputs and the previous outputs respectively. b is a vector containing a bias term for each neuron. At the first time step, the previous outputs are typically assumed to be all zeros.

Since there is one output for each neuron in the hidden layer, we need an additional fully connected layer followed by a softmax layer to convert this to class probabilities. In the case of sequence classification, where the goal is to map a whole sequence of inputs to a single class, we only need to consider the last time step, as this contains information passed on from all previous time steps. An RNN sequence classifier for three time steps is illustrated in figure 2.7

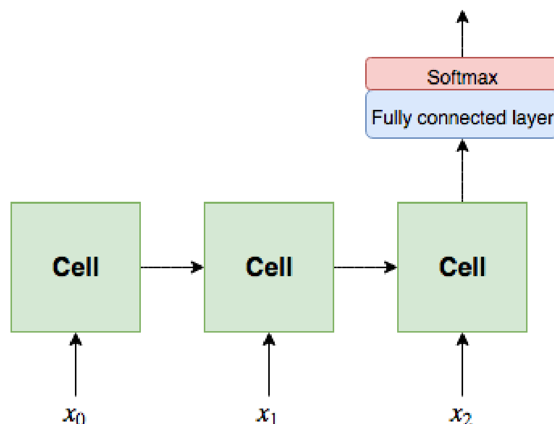


Figure 2.7: An unrolled sequence-classifying RNN.

For speech classification using MFCC data as inputs, each vector of 40 coefficients would correspond to a time step in the unrolled RNN. To classify a one second speech segment, the RNN would have 40 inputs at each of the 99 time steps, where the output at the last time step is converted to a confidence score for the wake word being uttered.

Alternatively, if a sequence of class probabilities corresponding to each time step is desired, the fully connected layer and softmax function can be applied at each time step. Typically, the weights and bias terms of the fully connected layer are shared across all time steps.

An RNN can be trained using a strategy called backpropagation through time, which is similar to the backpropagation method described in the previous chapter. Simply feed the network with some training data, evaluate the output using a cost function, and propagate the gradients of the cost function backward in time until the first time step is reached. Once the gradient of the cost function is computed with regards to each parameter in the network, this information can be used to update the parameters using gradient descent[9], or another optimization algorithm such as the Adam optimizer, which is an efficient extension to stochastic gradient descent with little memory requirement[10].

2.3.2 GRU Cell

If an RNN is trained on long sequences, the unrolled RNN will correspond to a very deep network. A problem which often occurs with deep networks is the vanishing gradients problem, in which the gradients get smaller and smaller as the backpropagation algorithm progresses down to the lower layers, leading to small weight changes and slow training. Alternatively, the gradients can

grow bigger and bigger, leading to large updates, which makes training diverge. This is called the exploding gradients problem. Additionally, due to to the transformations that the data goes through when traversing an RNN, some information is lost after each time step. After some time, traces of the first inputs will be insignificantly small. Because of these issues, the basic RNN architecture described in the previous subsection is rarely used in practice.

A better option may be using the Gated Recurrent Unit (GRU) cell introduced by Kyunghyun Cho et al. in 2014[11]. The GRU cell is a simplified version of the popular LSTM cell[1]. In order to solve the vanishing gradient problem, the GRU cell uses an update gate and a reset gate to decide what information should be maintained in the network. Thus, it is able to keep important information, and discard insignificant information. This process is learned through separate sets of weights. The equations for computing the output of a GRU cell are as follows[9]:

$$\begin{aligned} z_{(t)} &= \sigma\big(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)}\big) \\ r_{(t)} &= \sigma\big(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)}\big) \\ \tilde{h}_{(t)} &= tanh\big(W_{x\tilde{h}}^T \cdot x_{(t)} + W_{h\tilde{h}}^T \cdot \big(r_{(t)} \otimes h_{(t-1)}\big)\big) \\ h_{(t)} &= \big(1 - z_{(t)}\big) \otimes h_{(t-1)} + z_{(t)} \otimes \tilde{h}_{(t)} \end{aligned} \tag{2.15}$$

$z_{(t)}$ and $r_{(t)}$ are called the update gate and the reset gate respectively. The reset gate decides how much of the previous state should be forgotten, while the update gate decides how much of the candidate activation $\tilde{h}_{(t)}$ should be used in updating the cell state $h_{(t)}$.

The structure of a GRU cell can be seen in figure 2.8

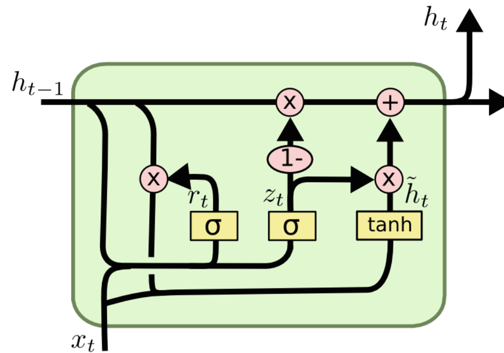


Figure 2.8: The structure of a GRU cell[1].

The GRU and LSTM cells are among the main reasons why RNNs have enjoyed so much success in recent years. As the GRU cell is simpler to implement and

understand, while seemingly performing just as well as its predecessor[12], it will be preferred in this thesis.

Chapter 3

Tools and Methods

In this chapter, the tools and methods used to produce the implementations will be presented. Two different systems are implemented using recurrent neural networks. The first one uses overlapping windows of fixed length to search for wake words in a continuous stream of audio. The second exploits the capabilities of recurrent neural networks in order to continuously listen for the wake word, removing the overlap and significantly reducing the number of computations needed.

3.1 Software Tools and Data

The implementation is entirely written in Python 3.5[13]. Python is an open source interpreted high-level programming language.

3.1.1 TensorFlow

TensorFlow 1.5[14] will be used to create neural networks. TensorFlow is an open source library used for numerical computation, which comes with strong support for machine learning and deep learning. Implementing a neural network in TensorFlow involves defining the network as a computational graph, and then running the graph inside a TensorFlow session.

A computational graph is composed of two types of objects: operations and tensors. Operations are the nodes of the graph, and they describe calculations that consume and produce tensors. Tensors are the edges of the graph, and they represent the values that will flow through the graph.

To evaluate tensors, a session must be instantiated. When you request the output of a node inside a session, TensorFlow backtracks through the graph and

runs all the nodes that provide input to the requested node. It is only during the session that the tensors of the computational graph actually hold values. Input values are typically fed to the graph using a feed dictionary, linking tensors or placeholders to the data which we want to input to the graph.

3.1.2 Other Modules and Libraries

The following list includes short descriptions of the external modules and libraries used for the project, as well as what they will be used for. It does not include modules from the Python standard library.

- `SciPy`[15]: a collection of open source software for scientific computing in Python. It includes, among others, the following packages:
  - The `SciPy library`[16]: a collection of numerical algorithms and toolboxes. Includes an io module which we will use to read wav files from the data set.
  - `NumPy`[17]: a package for scientific computing with Python. Its main object is the homogeneous multidimensional array, which we will use to store and perform computations on the data.
  - `Matplotlib`[18]: a plotting package which we will use to produce plots for this thesis.
  - `scikit-learn`[19]: a collection of algorithms and tools for machine learning. Includes a metrics module which we will use to evaluate the models.
- `python_speech_features`[20]: A library which provides common speech features for automatic speech recognition. We will use it to compute MFCC features for the raw speech data.
- `Keras`[21]: a high-level neural networks API. Includes some handy functions for pre-processing, including a function for padding variable length sequences which we will use.
- `PyAudio`[22]: provides bindings for the audio I/O library PortAudio. It will be used for audio recording in the real-time implementations.

3.1.3 Data Set

The data set used for this project contains 23957 recordings sampled at 48 kHz, where 5269 of the examples contain wake word utterances. 20 different phrases are uttered by speakers varying across different age groups and nationalities.

We divide the data set into three subsets: a training set, a validation set, and a testing set, with a distribution of 80 %, 10 %, and 10 % respectively. The data set is randomly shuffled before this split is performed, but the both the implemented systems use the same subsets for training, validation, and testing. All subsets share the same positive-to-negative ratio.

3.2 Sliding Window Approach

In this section, we design a wake word detection system based on a sliding window approach. Feature vectors extracted from a fixed-size sliding window are used as inputs to the neural network. The window needs to be large enough to contain the whole wake word utterance. Windows also need to overlap in time, so that no wake word utterance is split into two separate frames.

3.2.1 Pre-processing

The raw speech signal of each example in the data set need to be properly pre-processed in order to be suitable as input to a neural network. We want to produce a NumPy array containing MFCCs for each example, along with another array containing the corresponding labels; either 1 if the wake word is uttered, or 0 for any other utterance.

We choose a window length of 72000 samples, corresponding to 1.5 seconds at a 48 kHz sampling frequency. In order to generate more negatives, we extract two frames from each of the negative examples. The first is a randomly selected slice, and the second contains the loudest segment of the utterance. The loudest segment is found by sliding a window across the signal at regular steps, computing the energy of each window, and choosing the window with the highest energy.

For positive utterances, only the loudest frame is used. In both cases, examples shorter than the fixed window size are discarded.

A plot of a negative speech signal, along with two extracted frames, is provided in figure 3.1.

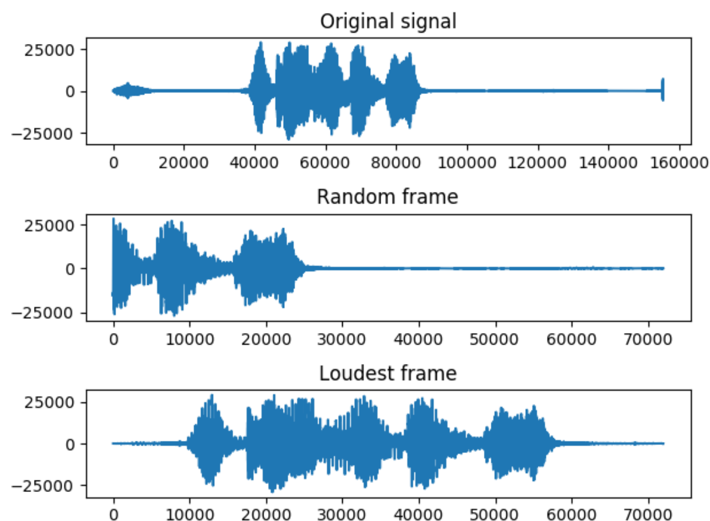


Figure 3.1: Original negative speech signal, randomly selected slice, and loudest frame.

MFCCs are computed for each frame using 25 ms windows (400 samples) with a 10 ms step size (160 samples). 40 coefficients are computed for each window using an FFT size of 2048. The mel-filterbank has 40 filters with the highest band edge at 8 kHz. The result will be a 149x40 NumPy array. Note that the python_speech_features module doesn't use zero-padding in MFCC computation. If it did, the dimensions would be 150x40.

Finally, the MFCC coefficients are normalized. This is done by subtracting the mean and dividing by the standard deviation of all the training data.

A positive utterance before and after pre-processing can be seen in figure 3.2.

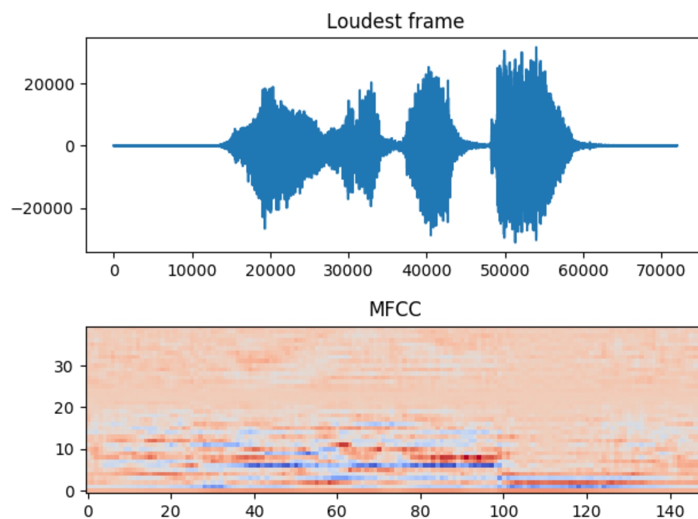


Figure 3.2: A positive utterance before and after pre-processing.

Pre-processed training, testing, and validation data, along with their corresponding labels, are stored in separate pickle files for later use. We also store the mean and standard deviation of the training data, so it can be used to normalize new data.

3.2.2 RNN Model Architecture

The MFCC matrices computed in the previous section can be viewed as sequences of 149 time steps, each holding a vector of 40 features. Thus, we can train a recurrent neural network to classify utterances based on this data.

We use a GRU network as described in subsection 2.3.2. As this is a sequence classification problem, we only consider the output produced at the last time step. Recall that this output holds information transferred through all of the time steps of the sequence. An overview of the unrolled model is provided in figure 3.3.

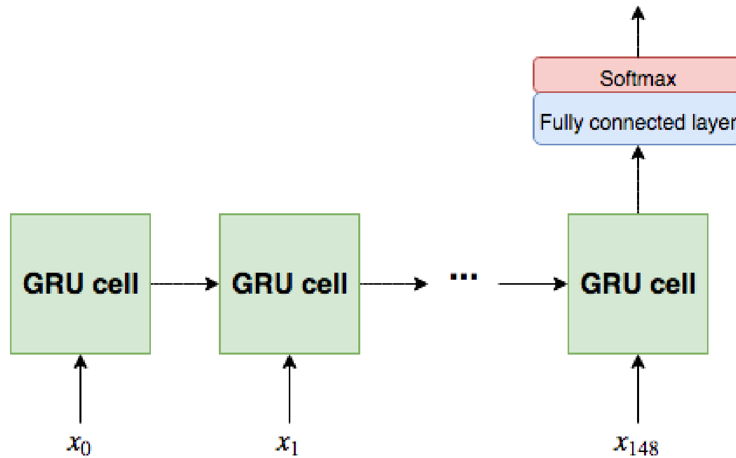


Figure 3.3: GRU model unrolled through 149 time steps.

We use a single-layer GRU model with 128 neurons in the cell. The fully connected layer at the final time step also has 128 neurons. In total, the model has 65,154 trainable parameters. The model is trained using batches of 128 training examples, with cross-entropy loss and Adam optimization, with a learning rate of 0.001. The whole training set is randomized at each epoch before it is divided into batches. A validation accuracy is produced after each epoch. We stop training the model after 20 epochs, as the validation accuracy is stagnating at this point.

3.2.3 System Implementation

After the model is trained, it can be used to detect wake words in a continuous stream of audio. The model is trained for windows of 1.5 seconds, and we use a step size of 0.25 seconds. Three consecutive windows of such a process can be seen in figure 3.4. In this case, we expect the third and final frame to produce a wake word detection, as it is the only one which contains the whole utterance.

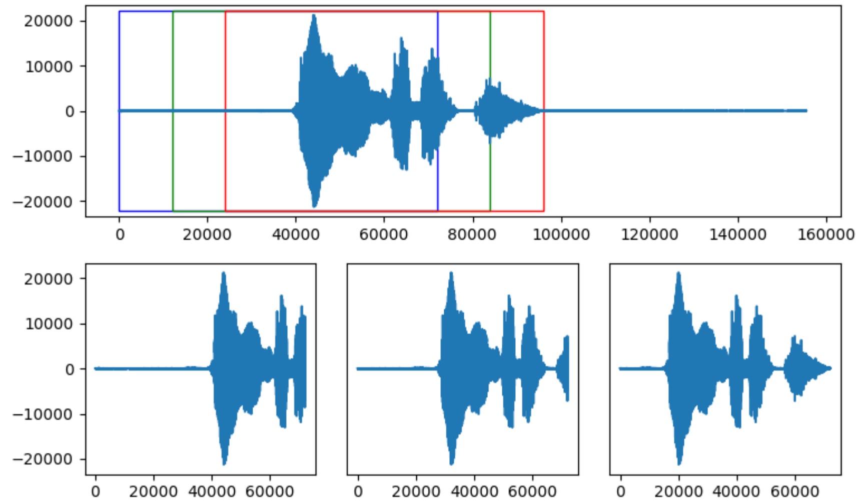


Figure 3.4: Three consecutive windows of a speech signal. The third frame contains the whole utterance.

In order to implement this in practice, we write a function which records 72000 samples at 48 kHz sampling frequency. For each window, we compute a 149x40 MFCC feature matrix. The feature matrix is normalized using the mean and standard deviation which we calculated from the training data. Then, the result is fed to the neural network, producing a probability score indicating whether the wake word was uttered or not within the recorded frame. If the score is higher than some pre-defined threshold, a detection is registered. The function is called recursively using threads every 0.25 seconds, meaning that overlapping frames are recorded and processed in parallel.

As mentioned, the window size has been set quite large to ensure that the whole utterance is contained within it. The downside of this is that multiple detections may occur based on a single utterance. To counter this, some smoothing of the sequence of produced probabilities may be performed. However, this has not been prioritized in this thesis.

3.3 Non-overlapping Window Approach

The sliding window approach is computationally inefficient, because of its need to use parallel threads which record and process overlapping data. It also requires the whole utterance to be contained within the pre-defined window length, which may not be suitable for slow speakers. Conversely, for fast speakers, the whole utterance may fall within consecutive windows, leading to consecutive

detections for a single utterance.

To counter these limitations, we design a different system which exploits the capabilities of a recurrent neural network. The basic idea is to use shorter frames, and to transfer the final state of a frame onto the first input of the following frame, instead of using a null state for the first input, thus eliminating the need for threads and overlapping computations.

3.3.1 Pre-processing

The pre-processing can be performed very similarly to how it was done for the sliding window based system. However, as we don't need a fixed window size, we can use the whole utterance as input without having to worry about varying sequence length.

Still, as we want detections to occur immediately after a positive utterance, we need to remove the trailing silence of each example. This can be done in the same way we used to extract the loudest frame in the previous section, but this time we keep the leading silent part. The resulting signal is shown in figure 3.5.

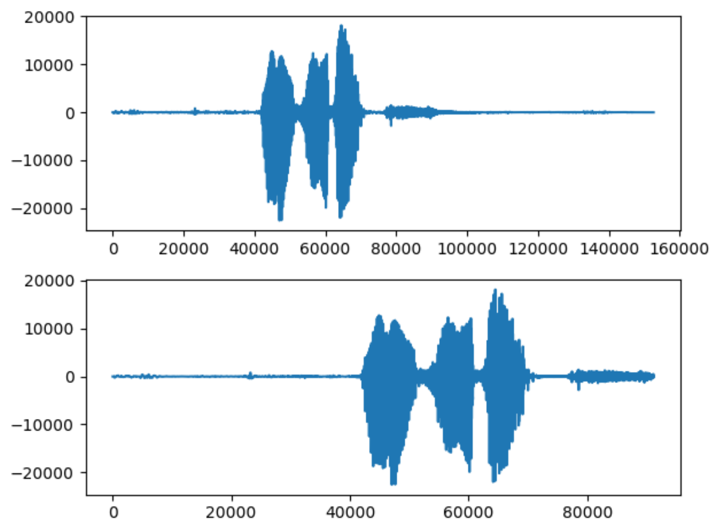


Figure 3.5: Speech signal before and after removing trailing silence.

Note that this is only necessary for positive utterances. For the negative examples, the whole signal can be used as input. However, to generate more negatives, we also extract one half of the utterance for each negative example.

Whether the first or the second half of the signal is used is chosen by a random process. This will produce two negatives per negative example, giving the same positive-to-negative ratio as the one used for the sliding window based model.

The MFCC computation, normalization, and data storage is performed in the same way as in the previous section.

3.3.2 RNN Model Architecture

The architecture of the model is quite similar to the one of the sliding window based model with a few very notable exceptions.

First of all, a wake word probability score is computed at every time step, which means that we need the fully connected layer with softmax activation at every output. However, only the last output is used to compute loss during training. This is because we want detections to occur only after the whole wake word has been uttered, and it would be difficult to find suitable labels for all other time steps. The other outputs are only used for real-time detection, as we want the flexibility of allowing detections to occur at any time within a frame. This is made possible by the fact that the parameters of the fully connected layer are shared across all time steps. The parameter sharing also means that this model will have the same number of trainable parameters as the model from the previous section.

Secondly, instead of using a null state for each new input sequence, we use the last state of the previous input sequence as the initial state of the current input sequence. During the training process, this means that if we use batches of 128 input sequences, the 128 final states of one batch will be used as initial states for the next batch. This is done by using a placeholder in the computation graph to store the initial state, and giving this the value of the last state at each iteration.

A basic overview is provided in figure 3.6. Note that the sequence length is no longer fixed to 149.

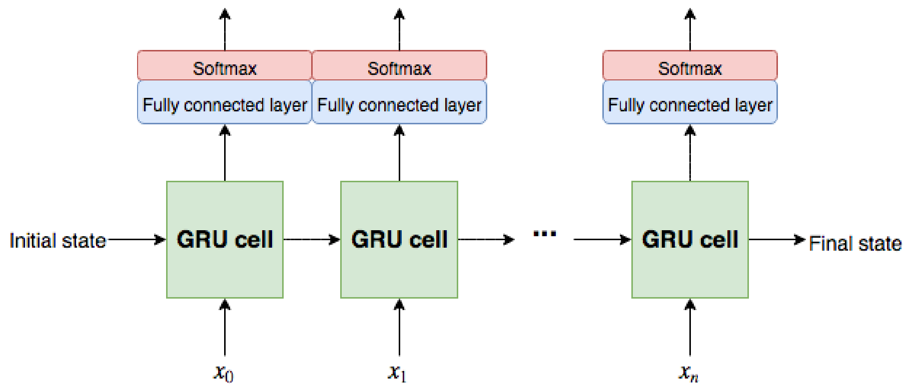


Figure 3.6: Overview of the model used for the system based on non-overlapping windows.

Training a TensorFlow model with variable length sequences within a batch is not trivial. First, we need to pad all sequences with zeros so that all of the sequences within the same batch has the same length. Then, at each iteration, we need to create a vector with one element for each sequence indicating its original length. This vector is passed to the model constructor as an argument, ensuring that the zero-padding does not affect the output and final state of the RNN.

Transferring the final states of each training batch onto the next one, as opposed to initializing each batch with a null state, means that instead of only considering the data from the current input sequence, the network will perceive that the current input sequence is just the trailing part of a much longer sequence. This, along with the fact that the sequence length is not fixed, means that the model eventually will learn to produce wake word probabilities at all time steps in a continuous stream of speech.

Otherwise, the model is trained in the same way as the model based on the sliding window approach.

3.3.3 System Implementation

The described system can be used to detect wake words in a continuous stream of audio more efficiently than the previous system. As there is no need for overlapping frames, we can use a step size equal to the frame size, which in practice can be of any value. However, due to the way MFCCs are calculated, there will be a small number of missing samples at the end of each frame. Thus, the frame size shouldn't be too small. Also, after producing wake word probabilities for a frame, we want to check whether any of the probabilities

exceed the chosen threshold. If the frame size is too big, it will delay this process. We choose to use 0.25-second frames, as in the previous section.

The wake word detection is a cycle of four steps: record a 0.25-second frame, pre-process the frame resulting in a matrix of 24x40 MFCCs, feed this to the neural network producing a vector of 24 wake word probabilities, and search for a probability exceeding a pre-defined threshold. If no wake words are detected, the final state of the frame will be used as the initial state of the next frame, allowing the system to continuously listen for a wake word. If a wake word is detected, the next frame will be initialized with a null state, resetting the memory and thus avoiding consecutive detections for a single utterance.

A basic overview of how the model is used to detect wake words in a continuous stream is illustrated in figure 3.7.

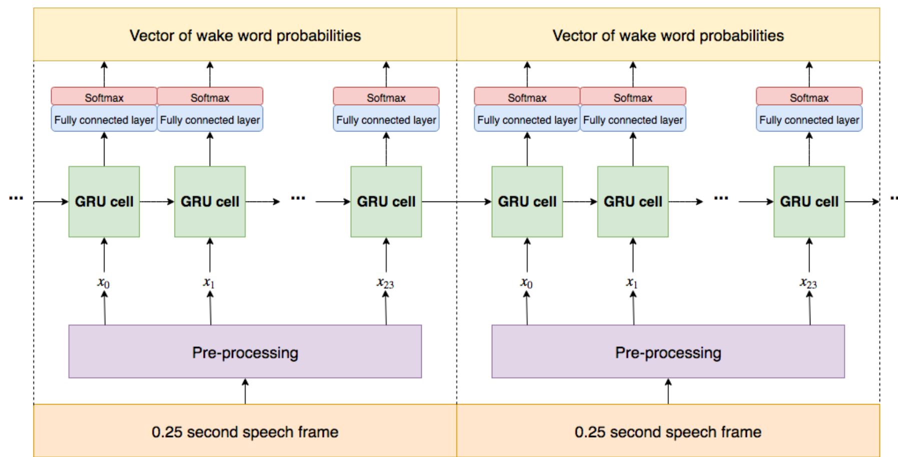


Figure 3.7: Continuous wake word detection overview.

Chapter 4

Results and Discussion

In this chapter, the systems designed in the previous chapter are evaluated and compared to each other.

First, we use the dedicated test subset to evaluate the models according to the performance measures presented in subsection 2.2.4. Be aware that the data set includes negatives which are very similar to the wake word. This means that false classifications may occur more frequently on the test data compared to other recorded data.

Then, we measure the CPU and memory usage of the real-time implementations of the systems described in subsections 3.2.3 and 3.3.3 respectively. We also apply the systems to longer audio streams, to get a sense of how they would perform in practice. We do this using four different files. Three of them are generated by choosing examples randomly from the test set and concatenating them. We use 20 negatives and 5 positives for each concatenated file. The systems are also applied to a podcast where we don't expect any wake word utterances to occur. The podcast used for testing is an episode of The Joe Rogan Experience[23]. It was chosen because it contains a natural dialogue between a man and a woman, and because of its length (2 hours, 38 minutes). It was originally a 44,100 Hz mp3 file, but it was converted to a 48,000 Hz wav file using SoX[24].

In the real-time simulation, the detection threshold has been set to 99 % for both systems, as this seems to work best in practice. The systems are run on a laptop with an Intel i5-4210u 1.7 GHz clock frequency dual-core processor.

4.1 Sliding Window Approach

The following results have been obtained using the sliding window approach. Recall that the test data was pre-processed along with the training data.

4.1.1 RNN Model Performance

Predicting the labels of the test subset using a detection threshold of 50 % gives an accuracy of 97.41 %. The confusion matrix is shown in table 4.1.

	Predicted class 0	**Predicted class 1**
True class 0	3611	47
True class 1	60	448

Table 4.1: Confusion matrix for the sliding window based model.

These numbers give a precision of 90.51 % and a recall of 88.19 %. This means that 90.51 % of the positive predictions are accurate, and when a positive prediction is uttered, it is correctly classified 88.19 % of the time.

Note that the number of false negatives is higher than the number of false positives. The number of false positives should ideally be low, as it is not desirable to wake up the system at unexpected times. By adjusting the detection threshold, we may get a higher precision, which in turn will reduce the number of false negatives.

Figure 4.1 shows how the precision and recall changes for different threshold values.

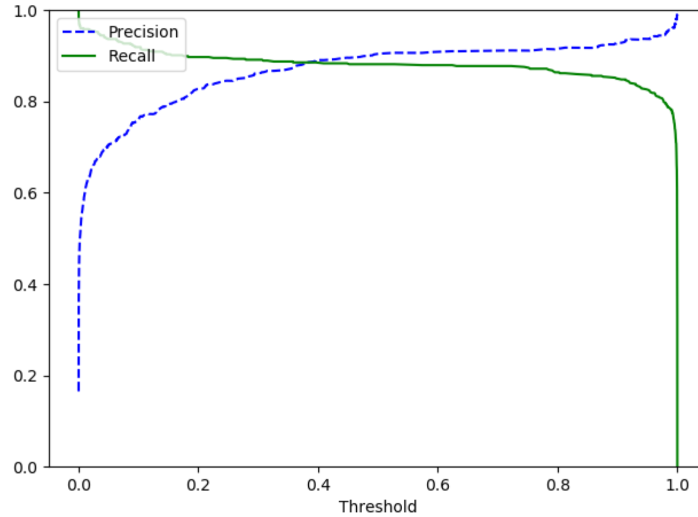


Figure 4.1: Varying precision and recall for the sliding window based model.

A selection of these values can be seen in table 4.1.1

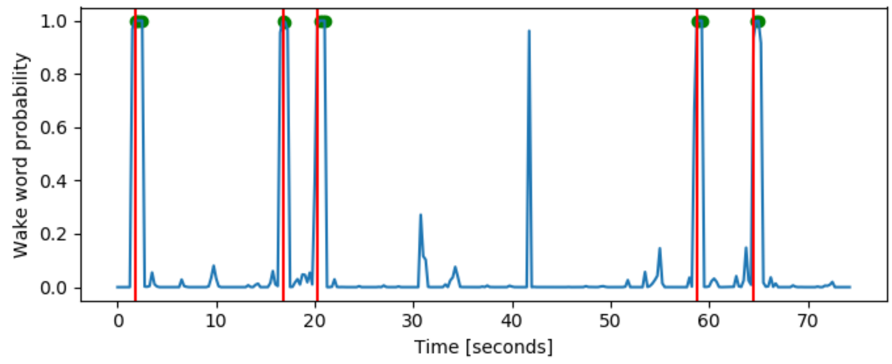
Threshold	**Precision**	**Recall**
50 %	90.51 %	88.19 %
75 %	91.17 %	87.40 %
90 %	92.89 %	84.84 %
95 %	93.74 %	82.48 %
99 %	95.89 %	78.15 %

Table 4.2: Precision and recall values for the sliding window based model.

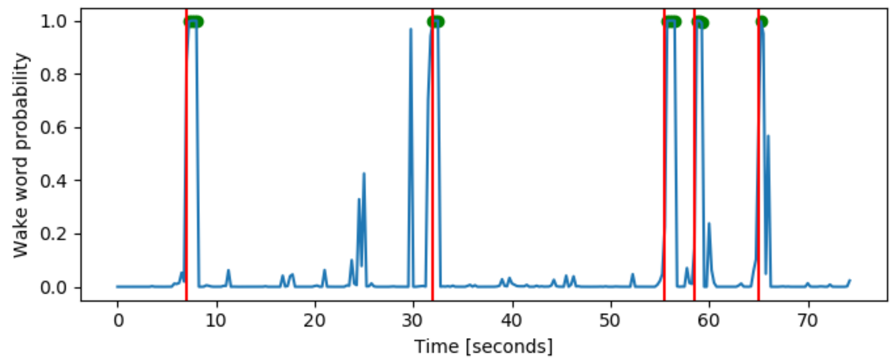
4.1.2 System Performance

Running the real-time implementation of the system utilizes around 6.7 % of the CPU on average while allocating 333,568 kB of memory.

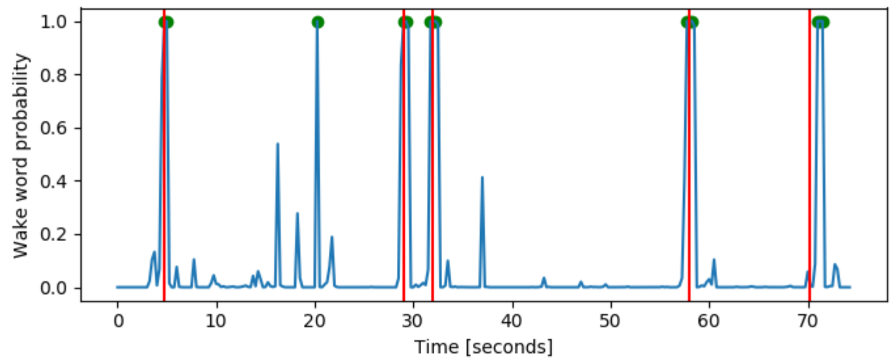
We can simulate the system by applying the same method to recorded files. We use 1.5-second windows with a step size of 0.25 seconds. As we only consider the last output produced for each window, we get one probability score every 0.25 seconds. A plot of the probability scores produced for the concatenated test files can be seen in figure 4.2. The red vertical lines show where a wake word has been uttered. The green dots indicate a wake word probability exceeding the pre-defined threshold.



(a) File 1



(b) File 2



(c) File 3

Figure 4.2: Plotted predictions from three files using the sliding window approach.

Apart from the false detection after approximately 20 seconds in the third file, the model performs as desired. This false positive, as well as any other high spikes seen in the plots, correspond to utterances that are very similar to the wake word.

Running the system on the podcast produces 18 false detections, which is a rate of 6.84 false detections per hour. Detections appearing in quick succession are assumed to be triggered by the same utterance and are thus not counted.

4.2 Non-overlapping Window Approach

The following results have been obtained using the non-overlapping window approach. The test data has been preprocessed in the same way as the training data. The model has been trained using non-null initial states for each example. For this reason, at the end of the final training epoch, the vector of final states was stored in a pickle file. For each of the test inputs, a state is chosen randomly from this vector and used as the initial state.

4.2.1 RNN Model Performance

Predicting the labels of the test set with a 50 % detection threshold gives an accuracy of 97.43 %. The confusion matrix is shown in table 4.3.

	Predicted class 0	**Predicted class 1**
True class 0	3619	39
True class 1	68	440

Table 4.3: Confusion matrix for the non-overlapping window based model.

These numbers give a precision of 92.05 % and a recall of 86.61 %. Figure 4.3 shows how the precision and recall changes for different threshold values.

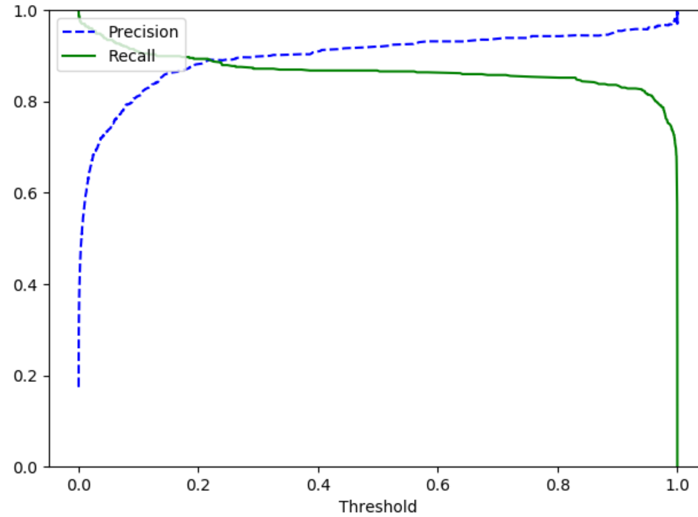


Figure 4.3: Varying precision and recall for the non-overlapping window based model.

A selection of these values can be seen in table 4.2.1

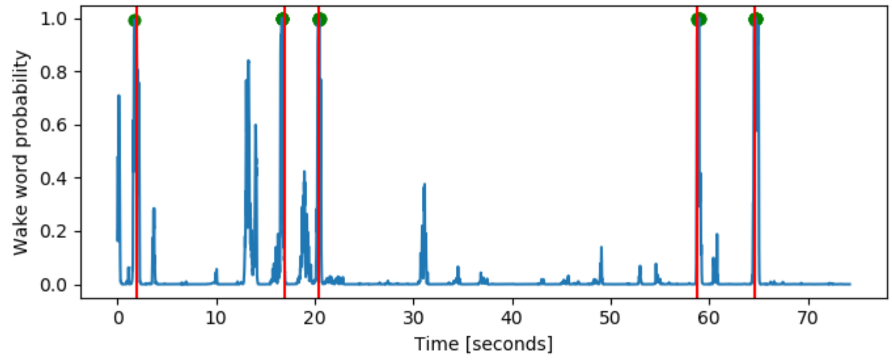
Threshold	Precision	Recall
50 %	92.05 %	86.61 %
75 %	93.94 %	85.43 %
90 %	95.50 %	83.46 %
95 %	95.84 %	81.69 %
99 %	96.92 %	74.41 %

Table 4.4: Precision and recall values for the non-overlapping window based model.

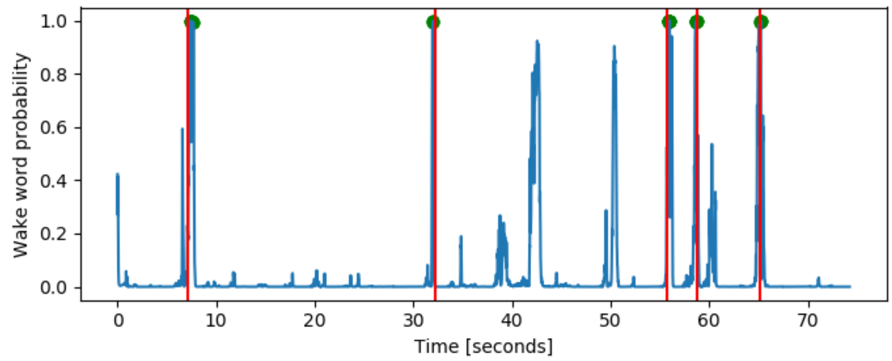
4.2.2 System Performance

Running the real-time implementation of the system utilizes around 1.4 % of the CPU on average while allocating 329,376 kB of memory.

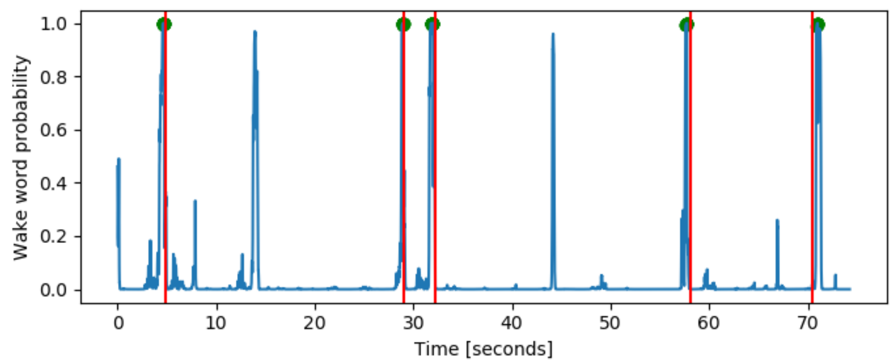
As there is no overlap in the windows, a wake word can occur at any time. Therefore we need to search through the outputs at every time step for a probability value exceeding the threshold. Plots of the probability scores produced for the concatenated test files can be seen in figure 4.4.



(a) File 1



(b) File 2



(c) File 3

Figure 4.4: Plotted predictions from three files using the non-overlapping window approach.

This system produces no false predictions at all for any of the three files.

Running the system on the podcast produces 17 false detections, which is a rate of 6.46 false detections per hour.

4.3 Comparison

The results obtained from applying the two RNN models to the test data set are very similar, with the test accuracy being nearly identical. This is not surprising, considering that the main difference between them is that the former is trained with fixed length input and null-state initialization, while the latter is trained with variable length input and state transferred from the previous input.

The results are summarized in table 4.3. Model 1 and model 2 refer to the model based on sliding windows and the model based on non-overlapping windows respectively.

	Model 1		**Model 2**	
Threshold	**Precision**	**Recall**	**Precision**	**Recall**
50 %	90.51 %	88.19 %	92.05 %	86.61 %
75 %	91.17 %	87.40 %	93.94 %	85.43 %
90 %	92.89 %	84.84 %	95.50 %	83.46 %
95 %	93.74 %	82.48 %	95.84 %	81.69 %
99 %	95.89 %	78.15 %	96.92 %	74.41 %

Table 4.5: Precision and recall comparison for the two models.

The main difference is that the first model generally has a higher recall, while the second model has higher precision. However, by using different threshold values for the two models, there are cases where the second model outperforms the first model both in terms of both precision and recall. For example, setting the threshold of the first model to 95 % gives 93.74 % precision and 82.48 % recall, while setting the threshold of the second model to 75 % gives 93.94 % precision and 85.43 % recall. This does not seem to be the case the other way around.

However, we observe an even greater difference in performance when comparing the implemented systems based on the respective models. An overview of this can be seen in table 4.3.

	Sliding window system	**Non-overlap system**
CPU usage	6.7 %	1.4 %
Memory allocation	333,568 kB	329,376 kB
False detection rate	6.84 per hour	6.46 per hour

Table 4.6: Comparison of the two implemented systems.

The false detection rates based on the downloaded podcast file are very similar, although the non-overlapping window approach has a slight advantage. Also recall that this system performed perfectly on the concatenated test files, while the other system produced one false detection.

As both the GRU cells and the fully connected layers share parameters across all time steps for both models, meaning that they have the same number of trainable parameters, there is no significant difference in memory usage for the two systems. However, while the sliding window approach uses 6.7 % of the CPU, the non-overlapping window approach uses only 1.4 %. Thus, the latter reduces the computational cost by nearly 80 %. This is mainly due to the lack of overlapping windows, which in turn reduces the number of computations it needs to perform drastically. It's also simpler to implement, as it doesn't require threading.

Chapter 5

Conclusion

5.1 Summary

The goal of this thesis was to design and evaluate a wake word detection algorithm based on deep learning. Two different recurrent neural network models were implemented. The models were trained using a data set containing speech utterances from a good variety of speakers. This data was pre-processed, and MFCC features were extracted and normalized.

The RNN models were used to implement two systems The first system uses a sliding window approach to search for wake words in a continuous stream. The second system exploits the memory capabilities of the recurrent neural network to continuously search for wake words without the need for any overlap. Both implementations were written in Python language using the TensorFlow framework.

The aim and expectation for the non-overlapping window approach was that it would perform similarly to the basic sliding window approach while significantly reducing the computation cost. Both models were evaluated using various performance measures, as well as a simulation of how the models would perform in a real-time application. Somewhat surprisingly, the experimental system ended up slightly outperforming the sliding window based system while reducing the computation cost by nearly 80 %. All in all, the results were highly encouraging, and showcase some of the capabilities and advantages of recurrent neural networks in speech processing.

5.2 Future Work

The results obtained are highly encouraging, though there is still room for improvement. Even after setting the decision threshold to 99 %, too many false detections occur when applying the systems on a speech based podcast. As the data set used is relatively small and limited, it is reasonable to assume that the number of false detection would decrease if the model was trained using more data and a broader range of unique utterances, including various types of noise and other non-speech sounds. In particular, the ratio of negative should be increased in order to give a more realistic distribution.

The model itself is also quite basic. Added layers, parameter tuning, and regularization techniques might lead to better performance. Also, it is possible to use other types of cells than the GRU cell, such as its predecessor, the widely popular LSTM cell. The low computation cost of the proposed model also allows for using ensemble modeling, in which confidence scores from two or more separate models are produced and combined.

Also, as the data set used for this project was sampled at 48 kHz, this sampling frequency was also used for audio recording in the real-time implementations of the systems. For speech, 16 kHz would probably be enough to capture the most important contents of the speech. This would further reduce the memory footprint and computation cost.

References

[1] Understanding LSTMs.
http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[2] Guoguo Chen, Carolina Parada, and Georg Heigold. *Small-footprint Keyword Spotting Using Deep Neural Networks.* Technical report, Center for Langauge and Speech Processing, Johns Hopkins University, Baltimore, MD, Google Inc., Mountain View, CA.

[3] Tara N. Sainath and Carolina Parada. *Convolutional Neural Networks for Small-footprint Keyword Spotting.* Technical report, Google Inc., New York, NY, U.S.A.

[4] Xuedong Huang, Alex Acero, and Hsiao-Wuen Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development.* Prentice Hall PTR, Upper Saddle River, New Jersey, 2001.

[5] Dony Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach.* Springer-Verlag, London, 2015.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[7] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning representations by back-propagating errors. nature*, 323(6088):533, 1986.

[9] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly, California, 2017.

[10] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization. CoRR*, abs/1412.6980, 2014.

[11] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. CoRR*, abs/1406.1078, 2014.

[12] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. *LSTM: A Search Space Odyssey*. *CoRR*, abs/1503.04069, 2015.

[13] Python 3.5 documentation. https://docs.python.org/3.5/.

[14] Tensorflow documentation. https://www.tensorflow.org/api_docs/python/.

[15] Scipy documentation. https://www.scipy.org/about.html.

[16] Scipy library documentation. https://www.scipy.org/scipylib/index.html.

[17] Numpy documentation. http://www.numpy.org/.

[18] Matplotlib documentation. https://matplotlib.org/.

[19] Scikit-learn documentation. http://scikit-learn.org/stable/documentation.html.

[20] python_speech_features documentation. http://python-speech-features.readthedocs.io/en/latest/.

[21] Keras documentation. https://keras.io/.

[22] Pyaudio documentation. http://people.csail.mit.edu/hubert/pyaudio/docs/.

[23] Joe Rogan podcast link. http://podcasts.joerogan.net/podcasts/candace-owens.

[24] Sox documentation. http://sox.sourceforge.net/Docs/Documentation.