BABEŞ-BOLYAI UNIVERSITY

CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

Romanian Speech Recognition Using Deep Learning

Supervisor

Lecturer Dr. Oneț-Marian Zsuzsanna

Author

Dolot Diana Nicole

2019

UNIVERSITATEA BABEȘ-BOLYAI
CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ÎN LIMBA ENGLEZĂ

LUCRARE DE LICENȚĂ

Recunoaștere vocală în limba română folosind deep learning

Conducător științific
Lector Dr. Oneț-Marian Zsuzsanna

Absolvent
Dolot Diana Nicole

2019

Table of Contents

List of Figures

Introduction

Speech recognition falls under the category of tasks that are very easily performed by most humans, yet are not at all as trivial for computers. However, digital assistants like Siri, Alexa and Cortana have definitely managed to achieve it proficiently through artificial intelligence and, as a result, the presence of speech recognition-powered applications in our daily lives is ever increasing. Personally, I have always been interested in the way these applications function and the actual software that powers them. I have also always been a bit disappointed in the lack of support most of these have for my native language, Romanian.

The goal I set when I embarked upon the task of working on this thesis was to explore what building an AI-powered speech recognition system entails and then to actually build my own. In my early beginnings, when reading about speech recognition algorithms, I have soon discovered recurrent neural networks, which seemed so interesting that I decided to try and experiment with them. Although an English speech recognition application would have been easier to develop, given the amount of English speech corpora available online and the abundance of other task-related resources, I have chosen the Romanian language on the basis of it being my mother tongue and a language I am genuinely fond of, and also exactly because of the challenges implied by the lack of resources. Both my findings and my actual work will be expanded on in the three chapters that follow.

The first chapter starts off the thesis by presenting to the reader the intricacies of the task of speech recognition. After defining the problem and providing further explanations about how one might represent the speech signal, it wishes to put speech recognition into perspective, by drawing a parallel between how it was achieved in its earlier stages and how it is tackled now, in some state-of-the-art applications.

The second chapter delves into the vast domain of deep learning. First, it defines basic concepts, such as supervised learning and artificial neural networks: their structure, but also how one can train and test them. Then, it moves on to define the particular type of neural network that is of interest to us in this thesis− the recurrent neural network. Finally, it presents a specific type of recurrent neural network architecture that this thesis makes use of, the long short-term memory neural network.

Finally, the third chapter presents the actual application that I have developed, based on the knowledge presented in the previous chapters. It starts from an overview of how it works from a user's point of view. Then it proceeds on explaining its underlying architecture, discussing the Keras framework in a more detailed manner, moving further to present implementation details, along with some key pieces of source code. This chapter will also discuss interesting challenges I have encountered along the way along with how (or if) I managed to overcome them.

Chapter 1

Speech Recognition

This chapter will introduce the notion of speech recognition and its relevance in today's world. Furthermore, it will discuss the ways in which the speech signal can be represented, with the aim of understanding the input one has when tackling the task of recognizing speech. Finally, a brief history of solutions to speech recognition will be laid out, with a focus on state-of-the-art systems, so as to understand better where the solution proposed in this thesis fits in the big picture.

1.1. Speech Recognition Today

Speech recognition is the process by which a computer maps an acoustic speech signal to text. It is also referred to as automatic speech recognition (ASR) and speech to text (STT). These terms will be used interchangeably throughout this thesis.

Presently, the most famous application of speech recognition technology are digital assistants. It is predicted that, by 2020, 50% of searches will be voice-activated [1]. One can observe usages of speech recognition throughout all aspects of daily life, from the workplace, with Alexa for Business recording minutes, scheduling meetings, initiating video conferences, to one's own home, which can be optimized with IoT speech-powered devices. Another area where speech recognition proved to be useful is language learning, where platforms like Duolingo and Busuu aim at improving their users' speaking skills by means of recording them saying certain phrases in their target language and then assessing the correctness pronunciation. Assistive technology, however, is the field where speech recognition makes the greatest difference. While specialized apps such as Voiceitt [2] provide a tool which translates speech that's not easily intelligible into clear sentences, even simpler dictation apps, such as Dragon Dictation, or even the built-in Chrome voice recognition system, can be of great help to people with disabilities which render their hands difficult to control.

1.2. Representing the Audio Signal

Upon embarking on the task of recognizing speech, before even trying to understand how it was carried out throughout history, it is crucial to understand what speech, and, more generally, sound is composed of and what tools there are for breaking it down into its building blocks.

1.2.1. Raw Audio Signal

The simplest (in the sense that it requires the least amount of processing, i.e. none) way of dealing with speech signal is just to use the original signal, unaltered, as can be seen in *Figure 1*. This approach has been tackled first in 2015, by Tara Sainath et al. [3], and then in 2018 by SincNet [4]. The main advantage of this approach is the fact that it does not require any further processing of the audio waveforms. On the

other hand, it implies storing a large amount of data and not all the information it provides is relevant to the task of speech recognition. Moreover some of this information might even make it more difficult. An example would be the background noise present in the raw audio signal, which some further processing might eliminate.

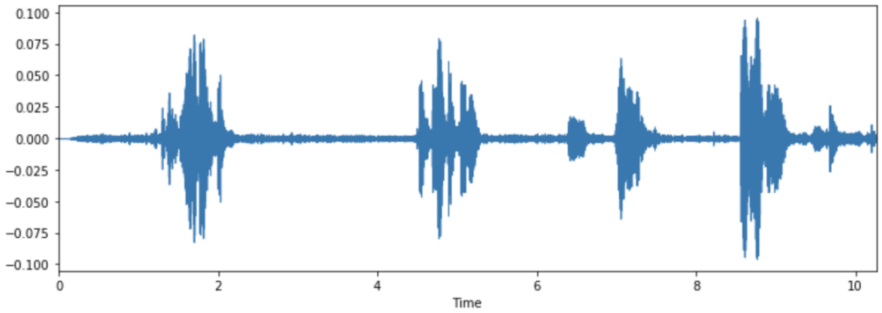


Figure 1. Audio Waveform

1.2.2. Spectrogram

A spectrogram is a way of visually representing the spectrum of frequencies of a signal as it changes with time and has always been one of the elemental tools for speech processing. It is typically depicted as a heat map, as can be seen in *Figure 2*, meaning the variations of energy are represented by different colors and changes in brightness. In the following figure, on the X-axis there's the frame number, while on the Y-axis there is the frequency.

The first step when creating a spectrogram is to split the audio signal in 20-40 millisecond frames. I will use 25ms frames for the purpose of the explanation. If shorter, there are not enough samples to get a right estimate of the spectrum and if longer, the signal changes too much during the frame and representing it becomes both a cumbersome and a fruitless task.

Sound is a vibration which propagates as an audible wave of pressure, through a transmission medium. In order to obtain such a spectrogram and considering the wave nature of sound, one begins by applying a *Fourier transform* on each of the 25ms frames of the audio signal, which is a mathematical function that breaks apart a sound wave into the simple sounds waves that compose it. By adding up the energy contained in each one of these individual soundwaves, one can determine a score which shows the importance of each frequency range. Obtaining that, one can color-code it and obtain a chart like the one in *Figure 3*. Considering that the chart belongs to the energies corresponding to the frequencies derived from a *single* 25ms window of audio, if one repeats this process, a spectrogram like the one in *Figure 2* is arrived at.

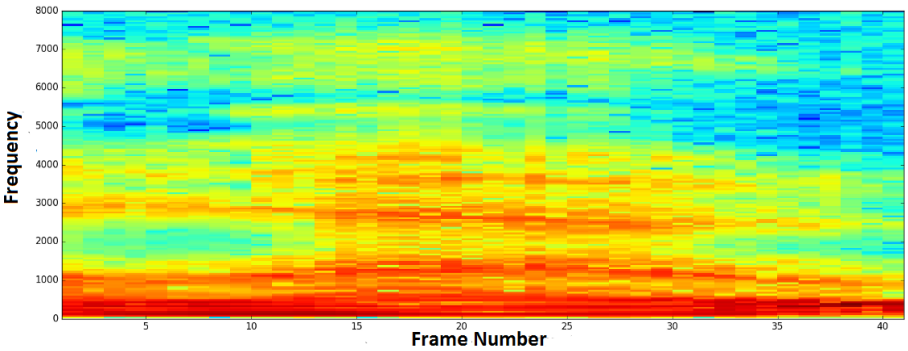


Figure 2. Spectrogram

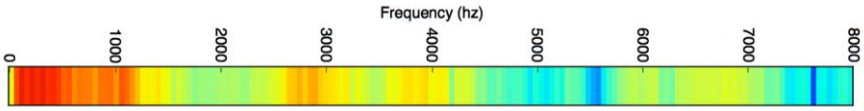


Figure 3. Frequency Chart Color-Coded by Energy

1.2.3. Mel Frequency Cepstral Coefficients

Similar to the spectrogram, the speech signal is split in 20-40ms frames and by applying mathematical transformations to these, one will obtain the vector of mel frequency cepstral coefficients corresponding to each frame. When boiled down to its essence, it is but a set of numbers representing the sound signal in the given frame. Out of the three ways of representing an audio signal presented in this chapter, this is the one that this thesis will be concentrating on. I will proceed with describing both how these coefficients are obtained and why they are of so much relevance to speech recognition.

Having computed the *power spectrum* for each frame, one proceeds with applying the *mel filter bank* to it, summing the energy in each filter, taking the logarithm of all filter bank energies, then taking the *discrete cosine transform* of the result and, finally, keeping the 2-13 resulting coefficients (out of 26). These are called the *mel frequency cepstral coefficients.* They were introduced in the 1980's by Davis and Mermelstein and ever since, they have been a central part of speech processing, representing the inputs for the majority of the algorithms aiming at recognizing speech.

To better understand the reasoning behind the process described above, the newly introduced terms will be defined:

- The *power spectrum* of a time series describes the distribution of power into frequency components composing the signal.
- In signal processing, a *filter bank* is a set of filters which divide the input signal into multiple subcomponents, each carrying a single frequency sub-

band of the original signal. What make the *mel filter bank* special is that it is based on the *mel scale* (depicted in *Figure 4*), which is a scale of pitches perceived by listeners to be equidistant from one another. Thus, it mimics the non-linear way in which the human ear perceives sound, in the sense that it can distinguish changes in lower frequency sounds better than changes in high frequency sounds. In other words, the human ear is more discriminative at lower frequencies than at higher ones and this is exactly the phenomenon the mel scales portrays. Integrating this scale makes our features simulate human hearing more closely. As can be seen in below, in the mel scale, a 1000 Hz tone is assigned a perceptual pitch of 1000 mels.

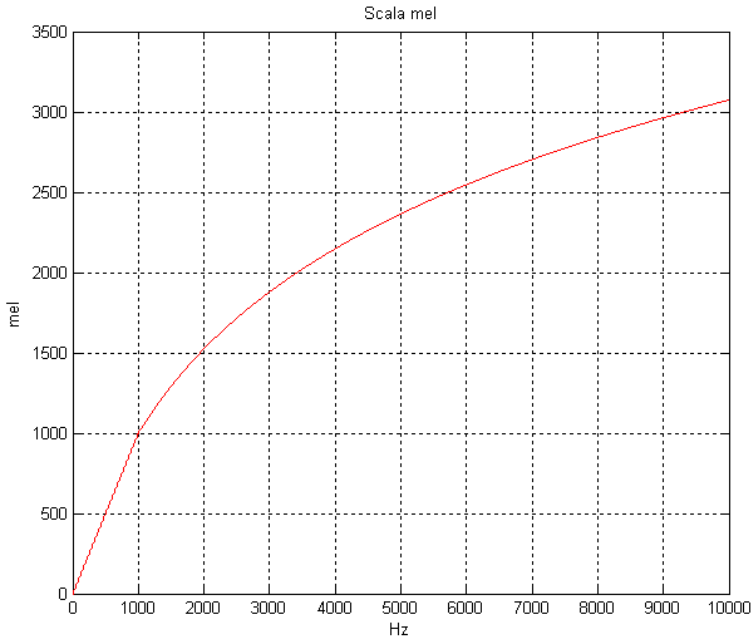


Figure 4. Mel Scale

The formula used for converting from frequency to Mel scale is:

$$M(f) = 1127.01048 * \ln(1 + f/700)$$

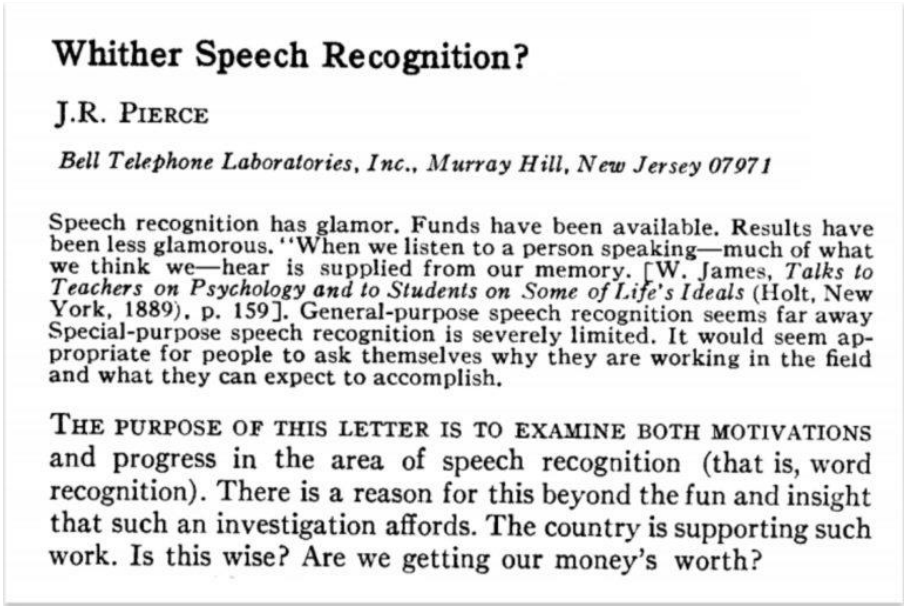
- Due to the fact that filter banks are overlapping, the filterbank energies are highly correlated to one another. The *discrete cosine transform* (in short, DCT) is applied exactly with the aim of decorrelating the energies.

It is to be noted that MFCC features are much lower dimensional than the spectrogram features, which is an advantage when one wishes to store them or subject them to computations.

1.3. Approaches throughout History

If now it is a billion dollar industry, valued at USD 9.12 billion in 2017 [5], and predicted to grow further in the following years (see *Figure 6*) with the likes of Alexa,

Google Assistant, Cortana and Siri, in the late 20th century scientists were asking themselves whether investing into speech recognition research was really worth it, as can be seen in *Figure 5*. Now let us explore how this area developed through time, both from the perspective of the products made available to the public, and of the technology underlying them, the latter being of the most interest to this thesis.



Whither Speech Recognition?

J.R. PIERCE

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07971

Speech recognition has glamor. Funds have been available. Results have been less glamorous. "When we listen to a person speaking—much of what we think we—hear is supplied from our memory. [W. James, *Talks to Teachers on Psychology and to Students on Some of Life's Ideals* (Holt, New York, 1889), p. 159]. General-purpose speech recognition seems far away Special-purpose speech recognition is severely limited. It would seem appropriate for people to ask themselves why they are working in the field and what they can expect to accomplish.

THE PURPOSE OF THIS LETTER IS TO EXAMINE BOTH MOTIVATIONS and progress in the area of speech recognition (that is, word recognition). There is a reason for this beyond the fun and insight that such an investigation affords. The country is supporting such work. Is this wise? Are we getting our money's worth?

Figure 5. Whither Speech Recognition? October 1969, The Journal of the Acoustical Society of America [6]

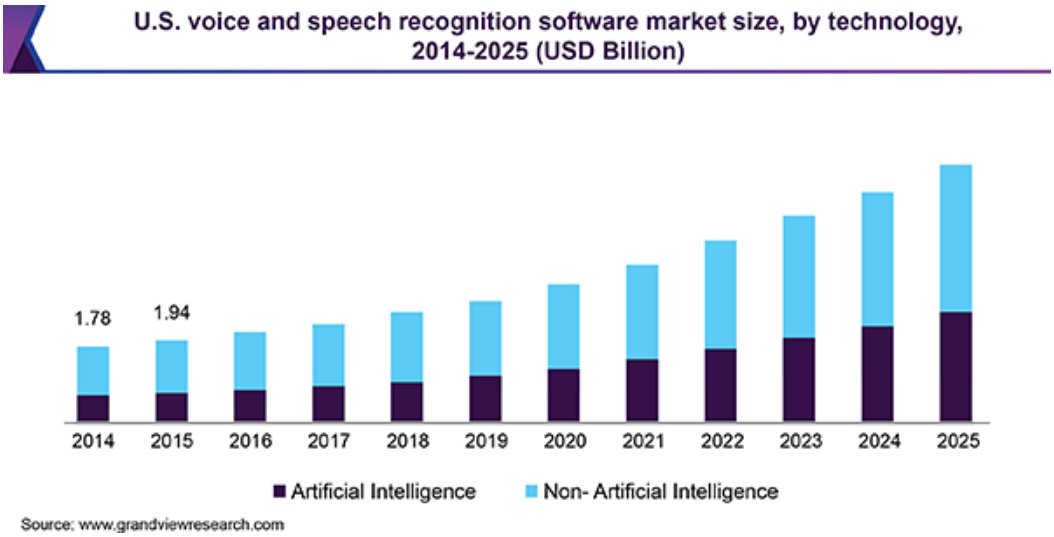


Figure 6. U.S. Voice and Speech Recognition Market Size 2014-2025 [5]

The year 1952 sees the emergence of genuine speech recognition with the invention of Audrey, at Bell Labs, a machine that filled an entire room and was able to recognize 0-9 digits with an accuracy of 90%, when those were spoken by its inventor H. K. Davis and an accuracy of 70-80% when uttered by a few other designated speakers. For other voices than those it was trained with, it performed considerably worse. As Clark Boyd

observed [7], this shortcoming of Audrey's hinted at a problem that is still persistent, to some extent, in the field of speech recognition− the great variation of speech stemming from characteristics of the speaker such as dialect, speed, emphasis and gender. Using the same technology, IBM's Shoebox managed to recognize 16 English words in 1962. Both of the two previously-described machines used pattern recognition methods, based on templates and spectral distance measure.

In the early 80s, the hidden Markov model (HMM), previously introduced by L.E. Baum in the second part of the 70s, became popular with speech recognition. A Hidden Markov Model is a probabilistic graphical model which allows for the prediction of a sequence of hidden (i.e. unknown) variables starting from a set of known variables. In other words, it facilitates the computation of the joint probability of a set of hidden states given a set of observed states. The first successful system built using HMMs was IBM's Tangora, which managed the remarkable task of predicting upcoming phonemes in speech and that could adapt to different voices. Tangora recognized the impressive amount of 20 000 words and even some full sentences.

Another leap in speech recognition happened in 1997, when Dragon released Naturally Speaking, the world's first *continuous* speech recognizer, which means that the user no longer had to utter each word separately. An upgraded version of it is still in use and on the market today. The technology stalled until 2008, when Google, making use of cloud computation, empowered speech recognition with a computational strength that was never seen before, which lead to the release of Google Voice Search for iPhone, one year later.

As Huang, Baker and Reddy observe, before 2010, a combination of HMM-based Gaussian densities have typically been used for state-of-the art speech recognition [8], until deep neural networks have been proven to be feasible and thus made popular by the milestone paper by Alex Graves et al., "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks" [9]. Although the concept of neural networks is as old as the other technologies used before it for speech recognition, only with the rise of the computational power of the beginning of the third millennium were they able to perform properly.

Throughout history, a particular framework (not to be read as a software framework, but more as a way of going about solving a problem; an underlying structure of a system) dominated the way people approached speech recognition. This is being referred to as *the traditional ASR pipeline* and can be seen in *Figure 7*. This method takes the audio signal and extracts relevant features (usually the mel frequency cepstral coefficients) which afterwards feeds as an input to the *decoder*, the *acoustic model*, the *lexicon* and the *language model*, all of them collaborating on determining the transcript of the initial audio signal.

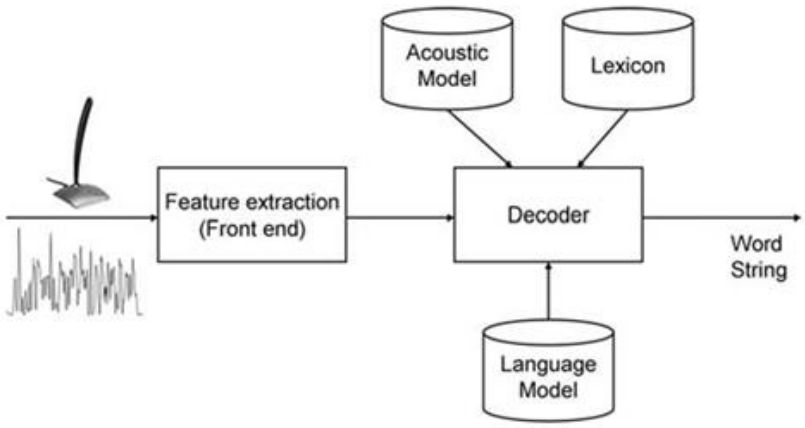


Figure 7. Traditional ASR Pipeline [10]

The *acoustic model* represents the relationship between the signal and the phonemes that make speech up, by creating statistical representations for each phoneme in a language, in the form of Hidden Markov Models. In short, the acoustic model maps acoustic features to phonemes. The *lexicon* (or pronunciation model), on the other hand, keeps track of words in the target language, along with their phoneme translations, while the *language model* contains the probabilities of a large number of word sequences and is very useful in disambiguating between similar acoustic ("let us pray" and "lettuce spray"). The pronunciation model is provided by a language expert, while the other models are both learnt through training. Finally, the *decoder* produces the output, the most probable transcript given the input audio, by means of approximate (greedy) search techniques.

Although traditionally speech recognition was being done on the basis of phonemes, it is undoubtable that this method implies significant overhead. Firstly, it requires a lexicon (as presented before), to keep track of all the phoneme translations of a significant number of words in a language. Secondly, it requires that the dataset in question provide phonemic transcriptions of the audio. Taking both of these into account, I have decided that my project will accomplish speech recognition on a letter basis and thus, from now on, this thesis will focus on this approach rather than the one involving phonemes.

Yet, regardless of the technique and audio representation being used, one thing stayed constant throughout time: the importance of the availability and amount of data. Since speech is a greatly variable signal and is described by numerous parameters, large corpora became central to modelling speech correctly and developing proficient systems. During the decades, we have seen the availability of recorded speech increasing at an exponential rate. Cloud-based speech recognition enabled the accumulation of huge amounts of recorded data, with Google and Bing having indexed the whole Web. This holds true for English, some of the most popular English speech corpora being TIMIT [11], Librispeech [12], WSJ [13]. When it comes to Romanian speech, however, I have only found one extensive corpus, namely the SWARA Speech Corpus [14], on which I will expand in the third chapter.

As a forecast, Huang et al. confidently assert that, in 40 years' time, speech recognition will pass the Turing test. But before delving into the opportunities that the future holds,

one should first comprehend the way state-of-the-art speech recognition is achieved nowadays.

1.4. State of the Art

When it comes to finding the information regarding the state of the art in speech recognition, a really useful starting point is wer_are_we [15], a GitHub repository attempting to track and centralize all such papers and applications which are on the bleeding edge.

However, before comparing performance in state-of-the-art applications, one should get acquainted with the tools used for evaluating how good a speech recognition system is doing.

1.4.1. Metrics for Measuring Accuracy

1.4.1.1. Word Error Rate

The most common, and a very straightforward metric, which is also used in wer_are_we is the WER (*Word Error Rate*). It takes into account the words predicted by the STT system, the target words and the differences between them. It is defined by the following formula:

$$WER = \frac{S + D + I}{N}$$

Here, S represents the number of substitutions, i.e. words that have been changed. D represents the number of deletions, meaning words that have been omitted; words that can be found in the target text, but not in the predicted one. I stands for the number of insertions (words that can be found in the predicted text, but not in the original), and N stands for the number of words in the target utterance (recording). It should be noted that there exist some variations to this formula, which assign different weights to the substitutions, deletions and insertions.

When comparing performance of different speech recognition systems, accounting for the word error rate does not suffice; it must be accompanied by the dataset on which the testing has been done, due to the difficulty that arises when comparing speech corpora. Therefore, there is no consensus on the most performant speech recognition system over all datasets. One can only talk about performant algorithms on a certain dataset. Thus, the most performant algorithm with respect to the LibriSpeech dataset [16] achieved a WER of 2.3%. The most accurate algorithm for the WSJ dataset [17] achieved a 2.9% WER, while the best algorithm for the Hub5'00 Evaluation Switchboard dataset [18] achieved a WER of 5%.

Interestingly enough, it has been claimed that humans are generally quite intolerant to transcriptions having word error rates higher than 15%, in the

way that for such error rates it is easier for a person to type a new transcript entirely from scratch than to correct it [19]. This claim has been made for optical character recognition, but it would make sense to apply to speech recognition as well.

1.4.1.2. Character Error Rate

Another metric for assessing the performance of an STT system is the CER (*Character Error Rate*). As can be deduced from its name, instead of putting emphasis on the words, as WER does, it centers on characters.

Character Error Rate is defined as the *edit distance* between the predicted text and the reference, divided by the number of characters in the latter. In turn, the *edit distance* assesses how different two strings are by counting the minimum number of operations required to transform one string into the other. There are different ways of computing the edit distance, the most popular being the *Levenshtein distance* [20]. The operations that this metric is based on are: insertion, removal and substitution, the same operations which can be observed in computing the word error rate. Therefore, the mathematical formula that defines it looks exactly like the previous one. Yet, the notations have different meanings. S is the number of substituted characters, D the number of deleted characters, I the number of inserted characters and N the total number of characters in the reference transcription.

$$CER = \frac{S + D + I}{N}$$

Behind the scenes, the way the computation is being done for both the word error rate and the character error rate makes use of dynamic programming

It is noteworthy that word error rates are usually higher than character error rates, and arguably so.

1.4.2. End-to-end Systems and Connectionist Temporal Classification

As can be observed in the wer_are_we repository, most of the state-of-the-art programs there are HMM-ANN hybrids, combining all the HMM learned throughout the last five decades and the newly acquired computational power, which enables the proper use of neural networks. With respect to the types of artificial neural networks being used, the LSTMs are by far the most popular. Nonetheless, there is an uprising trend towards fully neural network-powered (typically referred to as end-to-end) speech recognition systems. This was popularized by Baidu's Deep Speech paper [20] and this is also the direction my application will be taking.

The DeepSpeech solution is based on a recurrent neural network (which will be discussed in the following chapter), and when released in 2014, it managed to outperform all the previously published results on the previously mentioned Switchboard Hub5'00, by obtaining a 16% WER. It trained its RNN composed of 5 hidden layers on multiple GPUs and on 5000 hours of read speech from 9600 speakers. In order to eliminate plausible transcription mistakes produced by

homophone words, the developers of DeepSpeech decided to integrate a language model into their system.

As mentioned in the previous subchapter, an important tool for implementing end-to-end systems is the *Connectionist Temporal Classification* (CTC) algorithm and associated loss function, introduced by Alex Graves et al. in 2006. It is noteworthy that DeepSpeech made use of the CTC cost function as well for part of their training procedure.

The main issue with sequence problems where timing is variable, like speech recognition (but also handwriting recognition and action labelling in videos) is that, in the overbearing majority of cases, we do not know how the characters in the transcript and the audio align. This problem can be easily visualized in *Figure 8*.

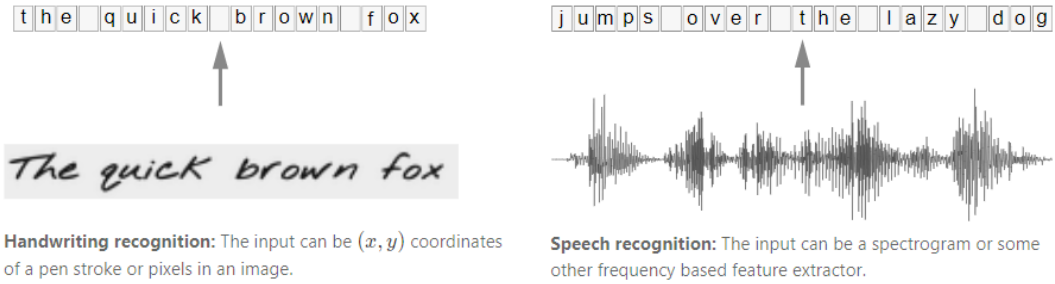


Figure 8. Sequence-to-sequence Tasks [21]

As Awni Hannun notes [22], one could try to work around this obstacle by:

a) Establishing a rule like "one character corresponds to five inputs" (i.e. to five 25ms frames of audio).
b) Hand-aligning each character to its location in the audio.

Unfortunately, a) can be easily dismissed on account of the fact that people's rates of speech vary−and not only from one person to another, but also for the same person, from one moment to the other, based on their emotions, the emphasis they want to put on certain words and multiple other factors. When faced with real-world datasets, b) fails as well, since the task of aligning every character to its location in the audio for every single utterance would be an extremely time-consuming task.

Reframing the discussion in a mathematical manner, the task of mapping speech to text can be translated to mapping an input sequence $X = [x_1, x_2, \dots, x_n]$ (the audio data; the frames, in our particular case) to an output sequence $Y = [y_1, y_2, \dots, y_m]$ (the transcript), where X and Y have variable lengths (n and m), whose ratios can vary. The issue identified previously is that of the lack of a correspondence of the elements of the two sequences.

Connectionist Temporal Classification is a type of neural network output and associated cost function (more on cost functions can be found in the following chapter), which comes as a solution to the issue of not knowing the mapping between the input and the output.

In the context of CTC, for each frame, a probability distribution over pre-established labels is outputted (by the underlying neural network). The set of labels varies with the target language. For Romanian, there are 33 possible labels: all the letters from a-z (including the special characters ă, â, î, ș, and ț), *space* or *blank*. The *blank* is a special character which is used internally by CTC, having no correspondence in the real world. After obtaining the most probable label for each frame, the following rules are applied in order to obtain the final prediction:

1. Merge repeated characters.

c	*c*	*blank*	*o*	*blank*	*p*	*i*	*blank*	*i*

⇨

c	*blank*	*o*	*blank*	*p*	*i*	*blank*	*i*

2. Remove any blank tokens.

⇨

c		*o*		*p*	*i*		*i*

3. The remaining characters represent the output.

⇨

c	*o*	*p*	*i*	*i*

From the previous example one can deduce the purpose of the blank label, which is allowing the transcription of words that have two (or more) of the same character in a row. It is noteworthy that the same word can be produced in numerous different manners, which models closely actual speech. For example, all the following sequences map to the word *copii* and are considered equivalent: ccc_opiii_i, co_p_i_i, ccooopppi_i_, __c_o_pp_i_i (where _ represents the blank label).

An important property of CTC can now be derived, which is that the length of the output sequence should always be less or equal than that of the input sequence. In other words, the alignment of *X* to *Y* is many-to-one. This can also be observed in the previous example, where the characters *ooo* in the input map to a single *o* in the output.

Taking into account the previously introduced mathematical notation, one can define the probability of obtaining the correct transcript *Y*, given the input *X*, as below. This probability, which can be seen as a score should be maximized.

$$p(Y|X) = \sum_{A \in A_{X,Y}} \prod_{t=1}^{T} p_t(a_t|X)$$

In the above formula, $A_{X,Y}$ is the set of valid alignments of a prediction that was derived from *X* that would give *Y* (following the previous example, *copi_i*,

*coooppi_i_* and *__c_o_pp_i_i* belong all to the set of alignments that, after decoded through CTC by applying the three aforestated rules, would output *copii*).

However, this is not the final CTC loss function. The actual function can be seen below and is the sum over the training set *D* of the minus logarithm of the previous conditional probability.

$$CTC(X,T) = \sum_{(X,Y)\in D} -\log p(Y|X)$$

Chapter 2

Deep Learning

This chapter will follow a top-to-bottom approach, introducing the general notion of Artificial Neural Networks, then moving on to a specific subtype of those, Recurrent Neural Networks, of which Long Short-Term Memory Neural Networks are yet another division.

2.1. Artificial Neural Networks (ANNs)

The term Artificial Neural Networks denotes a family of computational models mainly used for recognizing patterns, whose design and name are based on a simplification of the structure and behavior of neural networks inside the animal brain. Alternatively, in the words of the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen, they are computing systems made up of multiple simple, highly interconnected processing elements, that process information by their dynamic state response to external inputs [23]. They fall under the broad spectrum of Artificial Intelligence, where they fit into the Machine Learning paradigm. Therefore, they are adaptive algorithms that can perform predetermined tasks without being given specific instructions. Presently, they are being used for solving problems such as image recognition [24], machine translation [25], search engine optimization [26], medical diagnosis [27], playing games [28] and, most relevantly for the topic of this thesis, speech recognition [29].

2.1.1. Goal and Structure of an ANN

A standard Artificial Neural Network is a directed, weighted graph, whose nodes are called *artificial neurons*. The graph has a layered structure, with one input layer, one output layer and a variable number of intermediate layers, referred to as *hidden layers*. The flow of the data is from the input layer to the output layer. Nodes in the first layer do not have any predecessors, while those in the last layer do not have successors, and a node in a hidden layer can only be connected to nodes from its two immediately neighboring layers. A visual representation of a classic ANN can be seen in *Figure 9*.

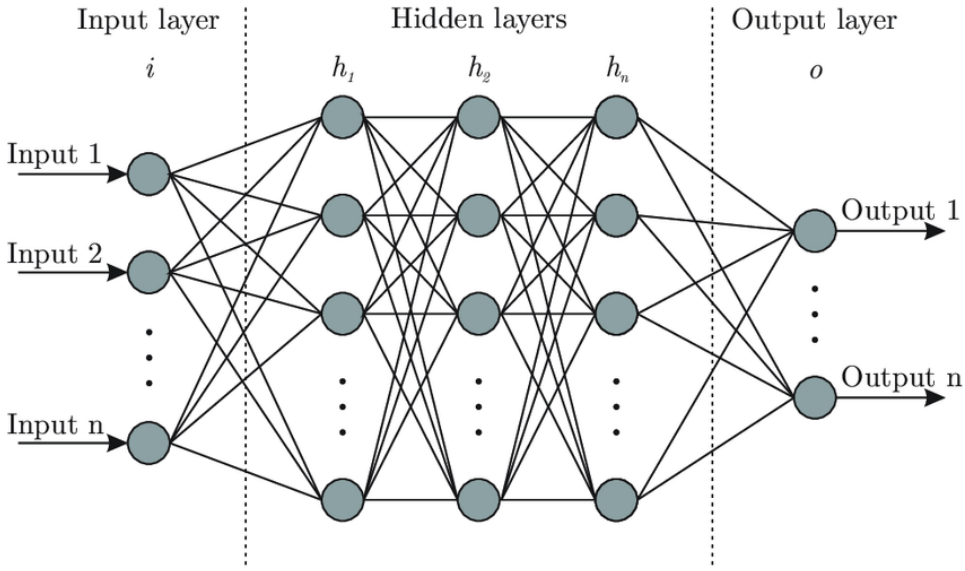


Figure 9. The Structure of an Artificial Neural Network [30]

The purpose of this architecture is that, starting from a certain set of inputs, and having a set of candidate outputs (i.e. possible solutions) to be able to point to the correct output. Now let us see how a neural network achieves the aforestated objective, using the previously-mentioned elements.

Zooming in on the artificial neuron, it is essentially a node which holds a value in the interval [0, 1], called *activation*.

As for the activations of the artificial neurons in the input layer, those are constant, in the sense that they represent external data. For example, they could be the values of pixels in an image, if the ANN is designed for image recognition, or certain features of sound waves, in the case of speech recognition tasks. Thus, their values are unchanging, with respect to a single data point (e.g. given a certain image, its pixels will always have the same values and the input layer of an ANN analyzing that image will always look the same, no matter the values held by the neurons in the network).

However, the case of the input layer is a special one and, in fact, it represents the only layer whose neurons' activations are independent of the rest of the network. Looking at a neuron that is not on the input layer (as seen in *Figure 10*), its activation is strictly tied to those of the neurons in the preceding layer and is given by the following formula:

$$f(b + \sum_{i=1}^{n} x_i w_i)$$

Here, x_i represents the activation of the i-th neuron in the previous layer, w_i is the weight corresponding to the edge between the aforementioned neuron and the one we are interested in, b is the *bias* corresponding to our neuron and f is the *activation function*. Intuitively, the *bias* represents the tendency of a neuron to be active or inactive and behaves much like a constant in a linear function.

$z(x) = b + \sum_{i=1}^{n} x_i w_i$ is called the neuron's *affine function*. The *activation function* maps the value of the affine function into the desired range.

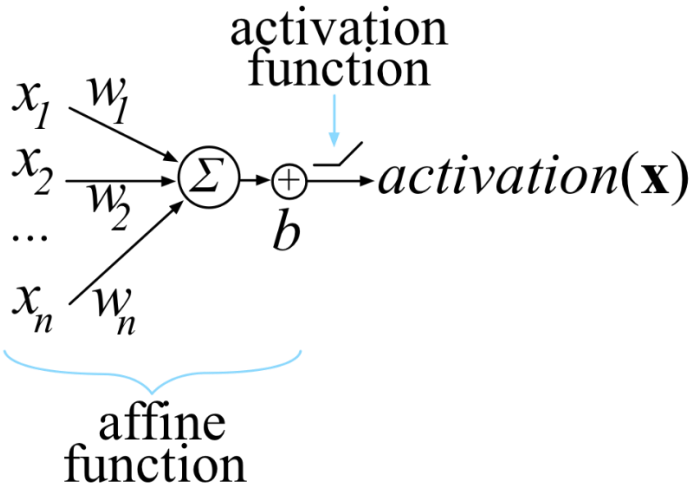


Figure 10. The Artificial Neuron [31]

2.1.2. On Activation Functions

Starting from the biological fact that neurons are electrically excitable cells [32], the activation function of an artificial neuron is an abstraction of the rate of action potential firing in a cell [33] (i.e. how excited the cell is).

I will proceed with briefly describing 5 of the most popular activation functions, which are useful in the scope of this thesis: *Heaviside step function*, *sigmoid*, *hyperbolic tangent* (*tanh*), *rectilinear unifier (ReLU)* and *softmax*, while keeping in mind that choosing the right one is highly dependent both on the task one has at hand and on the layer on which one applies it.

Heaviside step function, also called unit step function is a discontinuous function, whose value is 0 for negative inputs, and 1 for positive inputs, as seen in *Figure 11*.

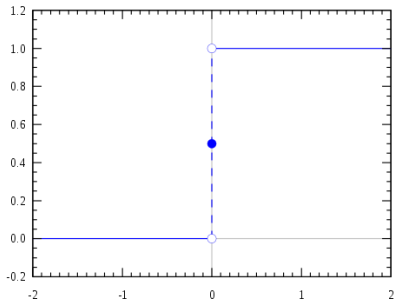


Figure 11. Heaviside Step Function [59]

The *sigmoid*, or logistic activation function is defined by the following formula:

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

As it can be observed in *Figure 12*, it maps all its inputs in the interval (0, 1). So, while the Heaviside step function tells us whether a neuron is active or not, the sigmoid gives information on *how* active that neuron is.

While it used to be popular, at present the sigmoid is mainly used for binary classification, in the output layer [34].

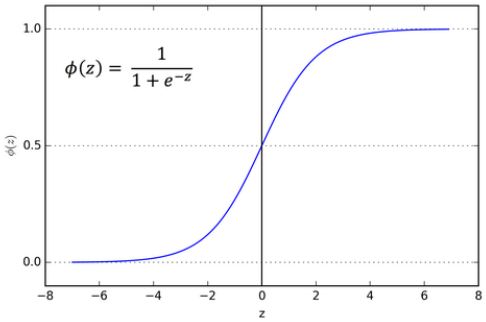


Figure 12. Sigmoid Function [35]

The *tanh* function is also S-shaped and maps inputs to the interval (-1, 1). Its formula is:

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

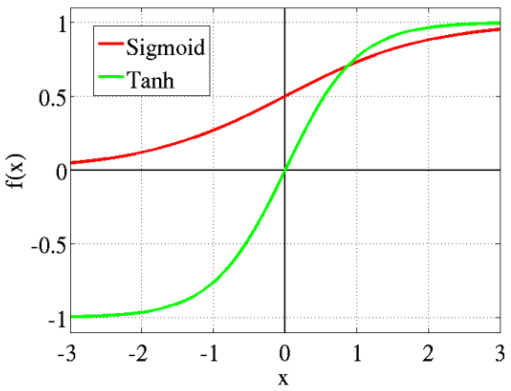


Figure 13. Tanh Function [35]

Its outputs are, thus, centered around zero and *tanh* is thought of as a scaled sigmoid. It is almost always chosen over the sigmoid function.

The *ReLU* function maps negative values to 0, and positive ones to themselves:

$$f(x) = \max(0, x)$$

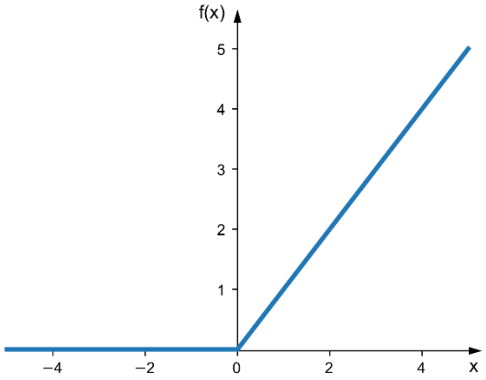


Figure 14. ReLU function

[36]

It is currently the default activation function for hidden layers [34].

Softmax is used in the output layer of a neural network and is an effective tool for classification. *Softmax*' utility resides in the fact that it takes into account the weighted sums and biases for all the neurons in the layer (also referred to as classes or logits), and, according to the formula displayed below, outputs a decimal probability for the given neuron (to be read as *class*).

$$p(y = j \mid x) = \frac{e^{(w_j^T x + b_j)}}{\sum_{k \in K} e^{(w_k^T x + b_k)}}$$

2.1.3. Training a Neural Network. On the Cost Function, Gradient Descent and Backpropagation

Now that the basis of the flow of data through a neural network has been laid down, it is time to answer the question of how a neural network, starting from a certain set of [input, desired output] pairs and randomly initialized weights and biases can learn to reach the right conclusion, given new inputs. Training and learning, in the context of neural networks, are used interchangeably.

Throughout this thesis, when using the term learning, I will be referring strictly to *supervised learning*, since this is the type of learning that is used in speech recognition. By supervised learning, we understand the act of teaching a neural network by means of labeled data (examples). That is, for example, when feeding the neural network a certain audio sequence as an input, the desired outcome (the word uttered in that particular sequence) will also be known and thus, at every step, we know whether the network's inference is correct (or rather how correct it is). In

unsupervised learning, on the other hand, the data is not labeled, making it harder to estimate how good the neural network is performing. As Adam Geitgey very nicely put it, supervised learning is like having the answers to the equations in a math test, but with all the arithmetic symbols being erased [37] (*Figure 15*). I would add, in a similar manner, that unsupervised learning is like looking at the math test, with the operators still being erased, but also without the answers (*Figure 16*).

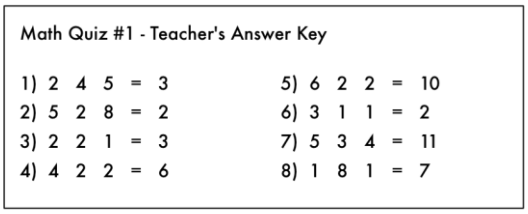


Figure 15. Supervised Learning Metaphor [37]

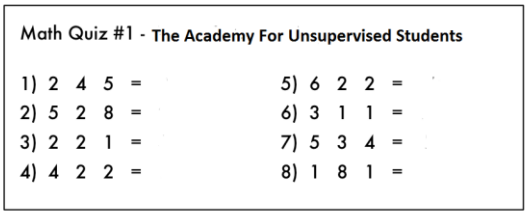


Figure 16. Unsupervised Learning Metaphor

In the most generic form, a neural network is just a function which maps an input to an output. By training it, we hope to achieve better outputs (i.e. outputs which are closer to the real ones). In order to obtain different, more preferable outputs, the inner workings (coefficients) of this function must be tweaked. But the only tweakable components of a neural network are the weights and biases. Therefore, training a neural network is synonymous with continually adjusting the weights and biases in order to improve the quality of its outputs.

Let us take a closer look at how this is being achieved. Firstly, whenever assessing the performance of a system, it is mandatory that there exists a criterion by which the evaluation is done. For NNs, this is called a *cost function* (or alternatively, a loss function). Given a labeled input and the output the network produces based on it, the cost function tells us how big a difference is between the network's prediction and the target value. It determines how well the network evaluates a particular entry. Choosing a cost function depends on the job being performed. One of the most popular functions of this type is the MSE (Mean Squared Error), defined below, where y is the vector of the labels, o is the vector of the network's outputs, and N is

the number of outputs (neurons in the output layer, classes). All this function does is squaring the difference between the outputs of the network and the actual values and computing the average.

$$C(y, o) = \frac{1}{N} \sum_{i=1}^{N} (y_i - o_i)^2$$

For the purpose of recognizing speech, however, I will be using another cost function: the CTC loss function, which was introduced previously, in the Connectionist Temporal Classification subchapter.

Nevertheless, no matter the choice of the cost function, the average cost of all training data represents the performance of the network, and the smaller, the better. Therefore, the problem becomes more mathematically tangible: in order to improve a neural network's performance, one should change the weights and biases with the purpose of lowering the value of the cost function.

The most popular algorithm for minimizing the value of a function is called *gradient descent*. In calculus, the gradient of the function determines the direction of the steepest increase of that function. Yet, our goal is to minimize the value of the function, therefore we must look in the direction where the function ascends, but in the opposite one, hence the term *descent*.

The gradient descent algorithm goes as follows:

1. Start at a random point. I call it random because it is dictated by the initial weights and biases, which are arbitrary.
2. Calculate the gradient of the cost function, which is composed by the partial derivatives of the cost functions, with respect to all the weights and biases. The resulting gradient will point in the direction of higher loss.
3. Take a step in the opposite direction from the one the gradient points at, which mathematically translates to updating the weights and biases according to the following formulas:

$$w_{t+1} = w_t - \eta \frac{\delta C}{\delta w} \text{ (for weights)}$$
$$b_{t+1} = b_t - \eta \frac{\delta C}{\delta b} \text{ (for biases)}$$

   w_{t+1} symbolizes the vector of the updated weights, which is obtained by subtracting from their initialize values the gradient of the cost function, denoted by *C* with respect to the weights, multiplied by the learning rate (η). The same goes for the biases. The learning rate is also an adjustable hyperparameter of the ANN, which controls how fast the weights and biases of a network are adjusted with respect to the loss gradient (as can be seen in the two formulas).

4. Repeat 2 and 3 until reaching either a global (best case scenario) or a local minimum (more probable).

It is important to point out that gradient descent does not guarantee finding the *global* minimum of a function, but it does assure a *local* minimum is reached.

At each iteration, the previously-described algorithm calculates the gradient after *all* the entries in the training dataset have been fed to the neural network. Because of this, the algorithm, which is also called *batch* gradient descent, can be very computationally heavy. Consequently, some lighter variations of it have arisen:

Stochastic gradient descent (SGD). Instead of going through each training example, it only uses one per iteration. Each iteration, the data is shuffled and a random example is picked. The most obvious disadvantage of this approach is that the path to the minimum point will be very hectic.

Mini-batch gradient descent is a compromise between the two other types of gradient descent. In each iteration, a predefined number of training data entries are processed at once. Though less noisy than SGD, this comes at a computational price.

A visual comparison between the 3 versions of gradient descent converging to a minimum can be seen in *Figure 17*:

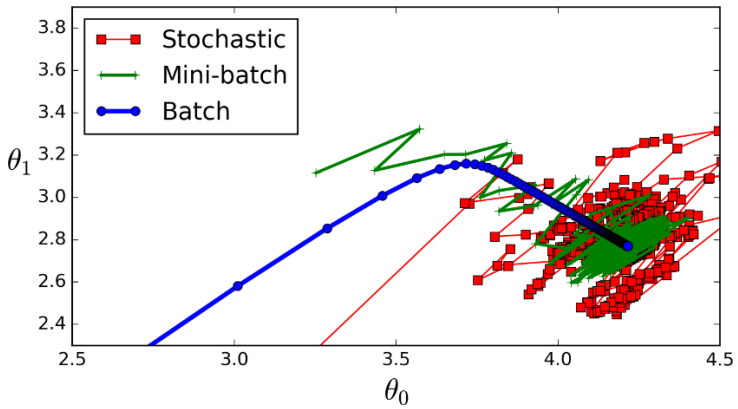


Figure 17. 3 Types of Gradient Descent [38]

In short, ANNs are trained by using the gradient descent algorithm, be it the original one or one of its adaptation. One of the prerequisites of using this algorithm is knowing the gradient of the cost function. In fact, this is obtain through another algorithm, called *backpropagation* [39], introduced in 1988, by David Rumelhart, Geoffrey Hinton and Ronald Williams. This essentially tells us how sensitive the cost function is with respect to each weight and biases, by moving backwards from the final error through the outputs, weights and inputs of every hidden layer.

2.1.4. Testing a Neural Network

Once the network has been trained and the desired performance has been achieved (evidently, on the training data), the ultimate proof of its capability is the testing

part. This stage of developing a NN consists in feeding it new inputs, data it has not been trained on. The measured accuracy of its outputs on this new, never-seen-before data represents the real accuracy of the system. Much like teachers should not test their students on the exact exercises they have solved during lectures, so are neural networks assessed using similar, but not identical data with the one they have been trained on. As students sometimes memorize whole solutions without really understanding the principle governing them (i.e. without actually learning), so can NNs "memorize" entire sets of data. This phenomenon is called overfitting and is not desirable.

2.2. Recurrent Neural Networks (RNNs)

The previously presented model of an artificial neural network is termed a *feedforward* network, due to the flow of the data through its nodes: always forward, from left to right, from the input toward the output. This design presents 2 main flaws which pose great difficulties when confronted with tasks such as speech recognition, image captioning, handwriting recognition, and speech synthesis.

First of all, a feedforward network treats all inputs as *being independent*. There is no persistence of its state from one input to another. When looking at speech recognition, it becomes clear how problematic this is, considering that sounds are not independent. For instance, when mapping sounds to letters, it is unrealistic to presume that the uttered letters have no connection to each other. They go on forming words and, for each language, there are probabilities that certain letters will be found next to each other in a word. Therefore, a structure that does not take into account the unbreakable bonds between letters will always fall short when trying to recognize speech.

Secondly, it imposes *too many constraints*. By design, a traditional neural network accepts a fixed sized vector as an input and produces a fixed-sized vector as an output. On top of that, the mapping is performed using a predefined amount of computational steps, meaning there is a fixed number of layers. This, too, is unsuited for tasks dealing with sequences, like the ones mentioned in the first paragraph, which involve either variable inputs, variable outputs or, as is the case for speech recognition, both.

Fortunately, Recurrent Neural Networks were specifically tailored to meet the two above-named needs and are the preferred type of neural network when dealing with sequences [40].

2.2.1. Structure and Memory

What sets RNNs apart from feedforward NNs and the characteristic that gives them their name is the fact that they have loops in them, as shown in *Figure 18*.

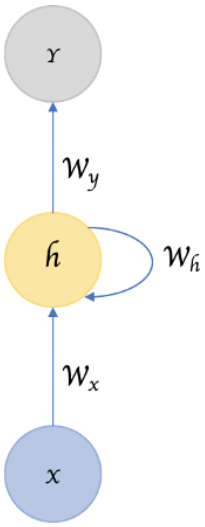


Figure 18. Recurrent Neural Network [41]

The presence of this loop which facilitates the ability of RNNs to reuse their output in the form of input is said to confer them *memory*. That is not to say that traditional neural networks do not possess a certain memory of their own. Still, these types of memory behave differently. On the one hand, the memory of feedforward structures develops itself only during training and can be observed in the way the network adjusts its parameters based on all the training data. The weights and biases can be seen as remnants of all the past data. After the training phase, however, the traditional network does not adapt further− its memory stops evolving. On the other hand, recurrent architectures presume the existence of a connection between events in a series, since they share the same temporal thread. Their entries follow a chronological order and with each entry, whether it is training or being tested, the network's memory transforms. A telling analogy which emphasizes the distinction between the two networks is presented in Skymind's article on RNNs and LSTMs [42] and goes as follows: As children, we learn to recognize colors and we use that knowledge for the rest of our lives anywhere we see colors, regardless of the context. That is analogous to memory in feedforward networks. It relies on an undefined past. A NN classifying colors has no interest in the data it was fed one minute ago, for it does not influence the current input being processed. Conversely, in our formative years we also learn to understand language, but no matter our age, the meanings we derive from a given sound are always contingent upon the sounds preceding and the ones succeeding it. Each step of the sequence is but another building block laid upon the previous one and their order gives rise to meaning. This resembles the memory of recurrent neural networks.

Now let us inspect how RNN's particular type of memory manifests itself. In order to elude the visual barrier posed by recurrence, we will consider the *unfolded* version of an RNN (*Figure 19*).

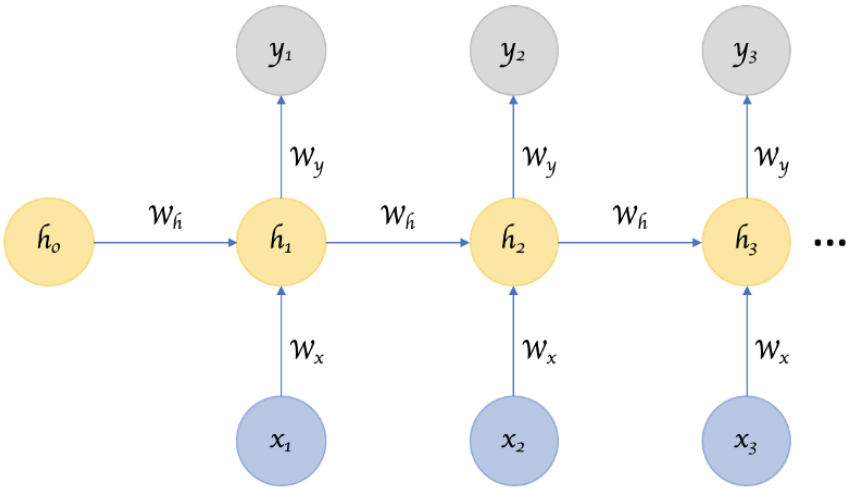


Figure 19. Unfolded Recurrent Neural Network [41]

Two equations denote the computations which are done at each time step in a simple RNN, like the one in the previous figure:

$$h_t = f(W_x * x_t + W_h * h_{t-1} + b_h)$$

$$y_t = g(W_y * h_t + b_y)$$

Notation-wise, x_t is the input, y_t the output, and h_t is the hidden node value, all at time step t. W_x represents the matrix of weights between the input and the hidden layer, W_h−the matrix of recurrent weights between the hidden layer and itself at consecutive time steps and b_y and b_h are the biases (which are not represented visually) and W_y−the matrix of weights between the hidden layer and the output layer. f and g are the activation functions of the hidden and output node, respectively.

Having the above image, the cyclic structure of the network can be reinterpreted as a deep network with one layer per time step and shared weights across time steps [43]. This perspective shows how RNNs, too, can be trained using backpropagation− more specifically, an extension called *backpropagation through time*, or BPTT [44].

2.2.2. Shortcomings

By the early 1990s, there emerged two major problems of the recurrent network architecture: the *vanishing* and the *exploding gradient*, which, as Michael Nielsen points out, have a common root and can be merged under the name of *the unstable gradient problem* [45]. This phenomenon has been first identified by Sepp Hochreiter in 1991 in his diploma thesis and then investigated extensively in 1994, by Yoshua Bengio, Patrice Simard and Paolo Frasconi [46].

As presented at 2.1.3., the gradient provides the impact that the change in weights and biases has on the total cost. It is calculated using backpropagation and used

afterwards by the gradient descent optimization algorithm in order to update the weights and biases.

The *vanishing gradient* problem appears in networks with a great number of layers and is defined by the values of gradients with respect to earlier layers becoming inconveniently (vanishingly) small. In case of the RNN, it occurs when dealing with long sequences. Given that gradient descent updates weights proportionally to their gradient, if the gradient is insignificantly small, then so will be the change in that particular parameter, leading to a weight that does not learn anymore. Such a stranded weight will hinder the learning ability of the whole network and make it impossible for the model to learn correlations between temporarily distant events. This translates to the loss of long term dependencies.

Let us look into how this problem comes to be. As stated earlier, the gradient of the loss function with respect to a certain weight is calculated by means of backpropagation. It is important to note that the backpropagation algorithm uses the chain rule in order to compute said loss, which, in essence, entails multiplying derivatives of the activation function. This implies that the earlier a weight is in the network, and the deeper the network, the more derivatives will be multiplied to calculate the gradient. The problem arises when all (or the majority of) these numbers are less than one, because the more such numbers one multiplies, the smaller the product gets. This can lead to a negligibly small gradient. Taking into account that to update a weight one must multiply the already-small gradient by the learning rate (generally less than 0.1), the weight will be updated by almost nothing, thus becoming stuck. The case of the exploding gradient is a similar one, but with big numbers being multiplied using the chain rule. Here, the weights change too abruptly and might never reach their optimal values.

2.3. Long Short-Term Memory Units (LSTMs)

In 1997, an improvement of RNNs, the Long Short-Term Memory Unit, was proposed by German scientists Sepp Hochreiter and Jürgen Schmidhuber [47] and it has been state of the art with respect to RNN architectures ever since. The unique architecture of an LSTM unit helps preserve the error that can be propagated through time and layers, thus permitting recurrent networks to learn over many time steps, enabling long term dependencies and combating the issue of the vanishing gradient. While inside a vanilla RNN unit there is typically a single layer (the activation function, most popularly *tanh*) −as seen in *Figure 20,* an LSTM unit is made up of 4 layers (the activation functions represented by the yellow triangles in *Figure 21*), interacting with each other with the aim of conferring it a reliable and distinctive long-term memory.

Let us analyze the reasoning behind the intriguing terminology. In simple RNNs (and ANNs in general), *long-term memory* manifests itself in the form of weights, since these keep updating during the training and enclose knowledge about the whole data having passed through the network. As regards *short-term memory,* it is present in the hidden

state passing from one entry to another [43]. However, the LSTM model introduces a third form of memory, via the architecture that will be detailed thoroughly in the following pages. The memory which is represented by these units is neither fully long-term, nor short-term, but rather an intersection of these two, hence the name *long short-term memory*.

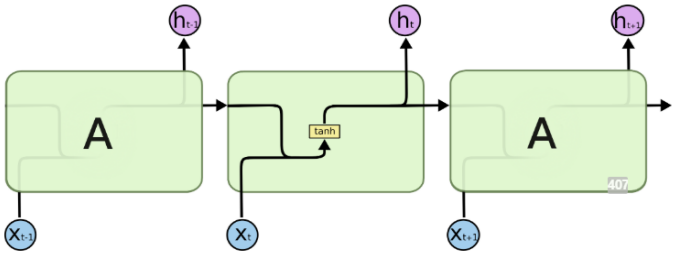


Figure 20. Standard RNN Units [48]

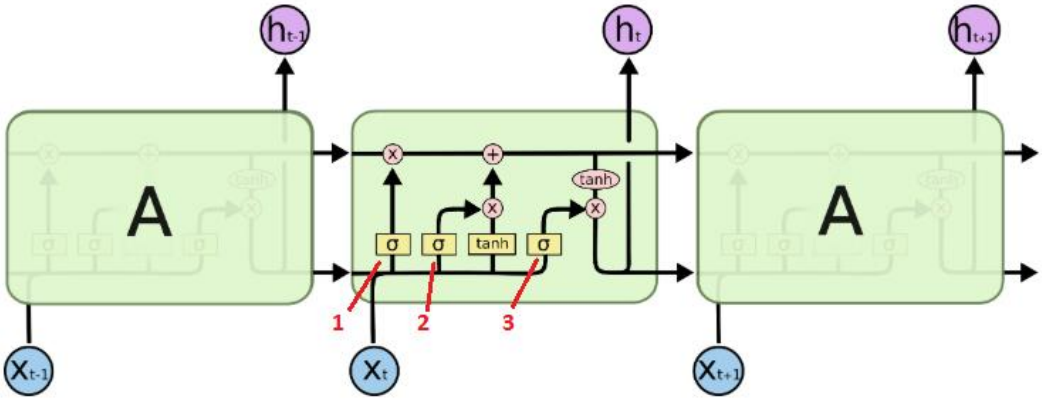


Figure 21. LSTM Units [48]

When addressing the structure of a long short-term memory unit and how all its components fit together, one should begin with the *cell state* (or *internal state*, as referred to in the original paper). It is visually represented by the bold horizontal line in *Figure 22* and it holds the *memory* of the network. It can be compared to a highway crossing the entire network and interacting only minimally with the processes happening within the cell (notice the 2 pink circles representing pointwise multiplication and addition).

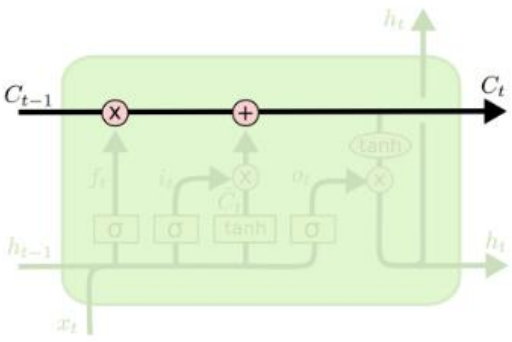


Figure 22. Cell State [48]

The LSTM has the power of changing the information carried by the cell state, through structures called *gates*. A gate is composed of a sigmoid function followed by a pointwise multiplication operation. As can be noticed in *Figure 21*, there are 3 such gates: the *forget* gate, the *input* gate and the *output* gate (marked 1, 2 and 3, respectively in *Figure 21*). Essentially, gates act as filters, or regulators. The connections entering and leaving these gates are weighted, as is characteristic of edges in neural networks. These weights are learned during training and determine how the gates behave.

To begin with, the *forget gate*, as its name suggests, decides the information that should be forgotten by the cell state. It takes as inputs h_{t-1} and x_t, the hidden state from the previous input and the current input. The sigmoid produces a value in the interval (0, 1) for each element in the cell state (denoted by the vector f_t). The closer this number is to 0, the more the element should be forgotten. Conversely, as it approaches one, the more important it is to remember it. As a remark, this gate was not part of the original design in 1997. It was subsequently introduced in 2000, by Gers et al. [49] with the aim of providing a method for the network to flush certain contents of its long short-term memory.

$$f_t = \Phi(W^{fx} * x_t + W^{fh} * h_{t-1} + b_f)$$

Second of all, the *input gate* decides what values in the cell state should be updated. Like the forget gate, it passes inputs the previous hidden state and the current input. The same input is passed, afterward, to a *tanh* function. In the original paper, this function was not tanh, but sigmoid. However, the tanh has been proven to be more effective, according to the state-of-the-art design introduced by Zaremba and Sutskever in 2014 [50]. While the sigmoid layer is responsible with the values that are to be updated, the tanh layer creates the vector of possible new values to be added for the state. Afterwards, the outputs of the two functions (i_t and g_t, respectively) are multiplied in a pointwise manner (obtaining p_t).

$$i_t = \Phi(W^{ix} * x_t + W^{ih} * h_{t-1} + b_i)$$

$$g_t = tanh(W^{gx} * x_t + W^{gh} * h_{t-1} + b_h)$$

$$p_t = g_t \odot i_t$$

After the two first gates (forget and input) have done their filtering, the cell state (C_t) gets updated, first by multiplying pointwise the forget vector, then by adding the previously computed product, p_t. (*Figure 22*).

$$C_t = C_{t-1} \odot f_t + p_t$$

Finally, the *output gate* tackles the new hidden state. Unlike the previous gates, it does not modify the cell state. The output gate passes the previous hidden state and the current input to a sigmoid function. Afterwards, the freshly updated cell state (C_t) is passed to a tanh function. Lastly, results of the two functions are multiplied and the hidden state (h_t) is updated accordingly.

$$o_t = \Phi(W^{ox} * x_t + W^{oh} * h_{t-1} + b_o)$$

$$h_t = tanh(C_t) \odot o_t$$

Since the original LSTM was introduced, many variations have appeared, which have also proven to be potent, such as the *gated recurrent units*, or GRU [51] and the LSTM with *peephole connections* [52]. However, at the basis of all those lies predominantly the previously described architecture.

Chapter 3

Application: Recognizing Romanian Speech Using Deep Learning

This chapter will discuss the Romanian speech-to-text application that I have developed. Starting from an overview of its functionalities from the perspective of the user, it will move on to a discussion on the architecture and the chosen frameworks. Finally, more insight will be given into the actual implementation details, along with the achieved results.

3.1. Overview

My final application consists of a desktop speech-to-text application, having a minimal user interface, powered by an underlying recurrent neural network and its interface can be seen in *Figure 23.*

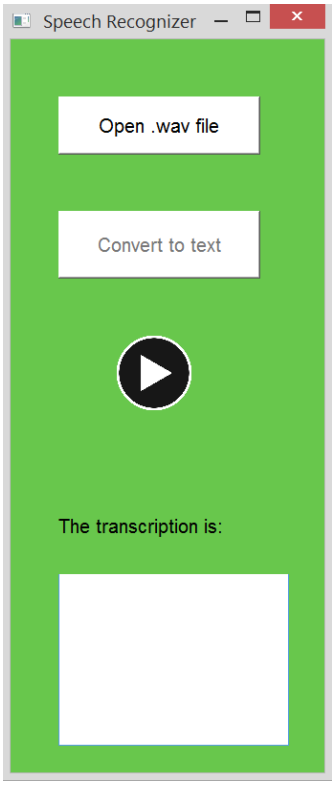


Figure 23. Screenshot of Application

The workflow of a user of this application looks as following:

1. Click the “Open .wav file” button.
2. Choose an audio file in the .wav format whose content one wishes to translate to text. After a file is chosen, then “Convert to text” button will be enabled.
3. (optional) Click the play button in order to listen to the chosen audio file.

4. Click the “Convert to text” button.

After step 2, the application will display the characteristics of the chosen file (length, bit rate) and after step 4, it will output the predicted transcription, which is also its main goal.

3.2. Architecture and Frameworks

Architecture-wise, I chose an object-oriented approach. Therefore, my application is divided into the following classes, which can be seen in the class diagram in *Figure 24.*

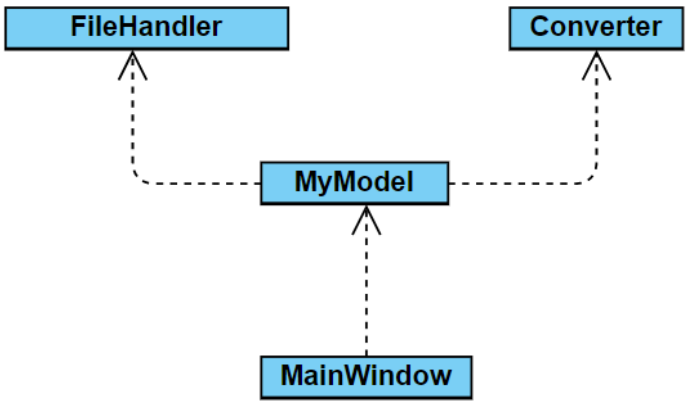


Figure 24. Simplified Class Diagram

The main classes are:

- *FileHandler* has to do mostly with reading and writing from/to the dataset files
- *Converter* deals mainly with conversions between strings and the project-specific integer representation of characters
- *MyModel* is a wrapper of the Keras Model, providing extra functionalities suited for the task at hand
- *MainWindow* is the class that encompasses the user interface of the application

The contents of these classes will be discussed in a more detailed manner in the subchapter tackling *Implementation Details.*

With regards to the technologies used, while the back end of my application is powered by Keras, the user interface was programmed using PyQt5.

3.2.1. Keras

Keras is an open-source neural-network library written in Python, which is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, Theano and PlaidML. As its primary author and maintainer François Chollet stated, it was conceived to be an interface rather than a stand-alone machine learning framework. My application uses Keras with TensorFlow as its backend engine.

Keras contains numerous implementations of commonly used layers, activation functions and optimizers. A feature of Keras that I found most useful for my project

is that, in addition to standard neural networks, it also has support for recurrent neural networks.

The whole library centers on the concept of a *model*, which represents the structure of an artificial neural network. There are two main types of models in Keras: the Sequential model and the Model class, which is used with the functional API. For this project, I have decided to use the latter. Whichever approach one chooses, the core functionalities are the same: one can add layers to a model (from a variety of predefined Keras layers), train it, test it and make predictions with it. The main difference between the two is that a Sequential model is a linear stack of layers, where each layer can only be connected to the one preceding it and the one following it, not allowing the creation of models that share layers or have *multiple inputs* or outputs, while the functional API allows for creation of more flexible neural networks. The two ways in which a model can be initialized are showcased in *Figure 25* and *Figure 26*.

```
from keras.models import Sequential
from keras.layers import LSTM

model = Sequential()
model.add(LSTM(32, activation='relu', input_shape=(200,)))
model.add(LSTM(10, activation='sigmoid'))
model.compile(loss='categorical_crossentropy', optimizer='adagrad')
```

Figure 25. Initializing a Sequential model with Keras

```
from keras.engine import Model
from keras.layers import LSTM

input_layer = Input(shape=(200,))
first_LSTM = LSTM(32, activation='relu', input_shape=(200,))(input_layer)
second_LSTM = LSTM(10, activation='sigmoid')(first_LSTM)
model = Model(inputs=input_layer, outputs=second_LSTM)
model.compile(loss='categorical_crossentropy', optimizer='adagrad')
```

Figure 26. Initializing a model using Keras' functional API

Although the two pieces of code written above have identical results, they stand to showcase the difference between the two approaches. In the first case, one just created the model by calling *Sequential()* and then adds layers to it, stacking them up on top of each other. In the second case, however, each layer is declared independently, mentioning the layer that precedes it (i.e. the layer from which it gets its input), e.g. *LSTM(…)(preceding_layer)*. When creating the model, one gives it an input layer (or multiple) and an output layer (or multiple). Having this information, the model includes all layers required in the computation of the output layer(s), given the input layer(s).

Be it functional or sequential, after creating a neural network model, one must *compile* it if one wishes to use it further. This method configures a model for training

and it requires at least an optimizer (could be the Stochastic Gradient Descent optimizer, which has been mentioned in the previous chapter) and preferably a loss function, which the model, when training, will try to minimize.

Here are the three key methods of the class Model that are relevant to this thesis (besides *compile*, which was discussed previously):

- *model.fit(train_data, train_labels, epochs, batch_size)*
  This takes the training data, as a numpy array, and the labels of said data, in the same format, and trains the compiled model (adjust its weights) over a given number of epochs, using the given data. An epoch is a full iteration over the entire data provided. The data may also be split in batches, in which case a batch size is also given to the method. It returns a History object, containing a record of the loss values achieved throughout the training.

- *model.evaluate(test_data, test_labels, batch_size)*
  This method takes the test data and the corresponding labels, both as numpy arrays and potentially a batch_size, if one wishes to conduct the testing in batches. It returns the recorded loss value.

- *model.predict(data, batch_size)*
  This method generates output predictions for the given input data. It returns the predictions in the form of a numpy array.

All of the above methods have additional optional arguments, but since I have not used them while developing my model, they are not very relevant to the subject of this thesis, so I have decided to omit them.

3.2.2. PyQt5

Because the user interface was not my main focus, I just needed a functional one for a desktop application. So I chose PyQt5 based on my familiarity with the Qt framework and the Qt Designer from university (albeit for C++) and recommendations from peers who have used it before for their projects.

I have used the Qt Designer (as shown in *Figure 27*) for arranging the visual components.

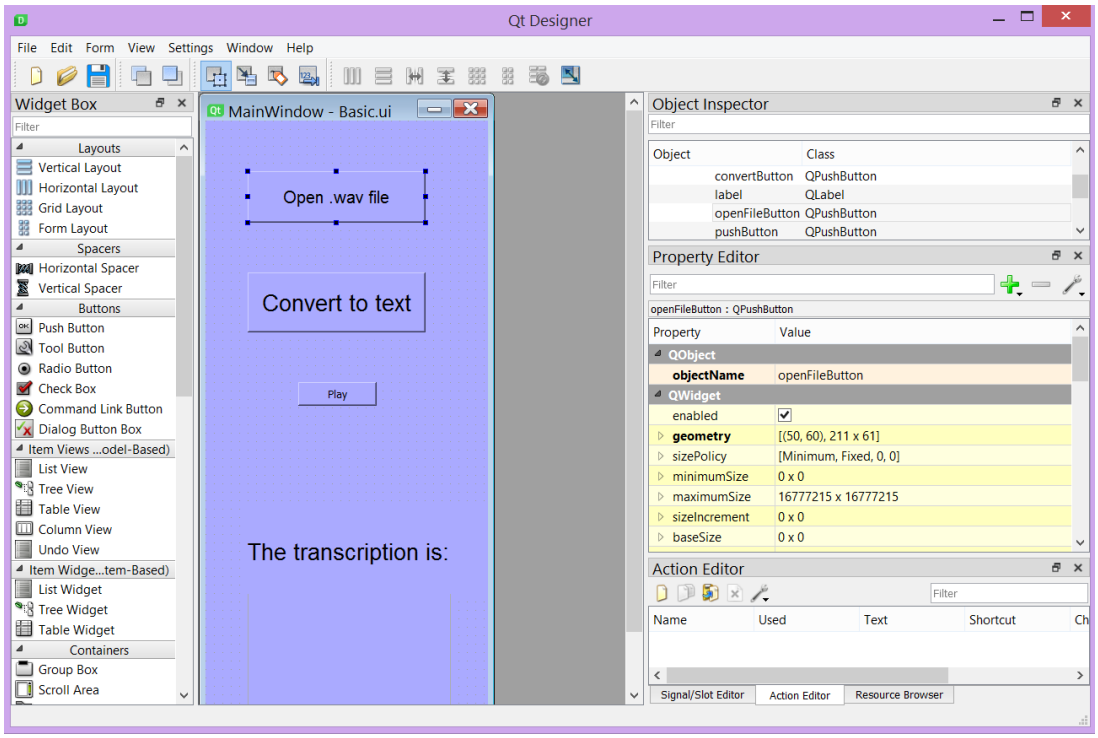


Figure 27. Qt Designer

Then, I generated the corresponding python code, using the pyuic5 command-line tool. I did so by running the following command:

Pyuic5 –x "AppUi.ui" –o "AppUi.py"

However, the generate code needed refactorization in order to be readable and for me to be able to work with it, so I ended up rewriting a big part of it. Nevertheless, Qt Designer still helped with the coordinates of the buttons, labels and the other visual elements, which otherwise would have been burdensome to find by means of trial and error.

3.2.3. Google Colab

Google Colab is a free Jupyter notebook environment provided by Google where one can use GPUs and TPUs for free for computational purposes. It has been of great help in the process of training my neural network, by providing the computational power that my laptop lacks. Thus, I could train the neural network online while I wrote my thesis on the laptop without running the risk of my laptop overheating or shutting down, leaving me with unsaved work and a half-trained neural network. In all fairness, this is half true, because unfortunately, from time to time, my Google Colab would disconnect and stop my code from running. The way I used Google Colab was the following: I would train the network on the platform and save the obtained weights in an .h5 file. Then I would use the network (i.e. its corresponding .h5 file) that I had deemed the best in the final program, the one on my own machine (the one with the integrated UI).

3.3. Dataset

The first challenge one faces when taking up the task of building an ASR system for the Romanian language is finding a big enough speech corpus, one with as many hours of recordings, produced by as many speakers as possible. Unfortunately, as recorded by MetaNet in "The Romanian Language in the Digital Age" [53] and as observed in *Figure 28*, the Romanian language has fragmentary support when it comes to language technology resources.

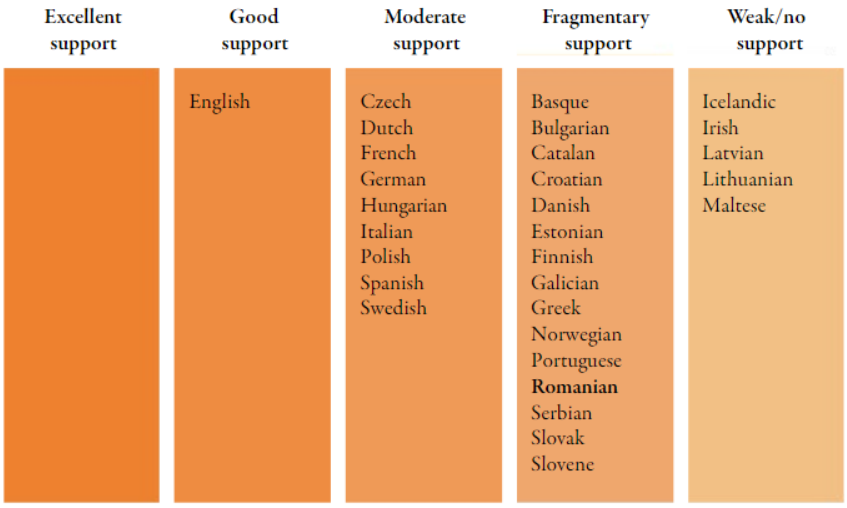


Figure 28.Speech and Text Resources: Existing Support for 30 European Languages [53]

The dataset which was used for training and testing the neural network is the SWARA Speech Corpus [54], which consists of 21 hours of recordings from 17 speakers, 9 female and 8 male. It amounts to a total of 19 279 utterances, sampled ad 48kHz sampling rate and is accompanied by the orthographic transcripts and semi-automatic phone-level alignments (which would serve a phoneme-based approach), of which only the former has been used for the purpose of developing this application. The speakers were aged between 20 and 35 years old at the time of recording and they had no self-declared hearing or speaking impairment. Their regional accents are mild.

The only other bigger Romanian speech corpus that I found is *CoRoLa−The Reference Corpus of Contemporary Romanian Language* [55], which has 151 hours of recordings. It is at least 7 times larger than the SWARA dataset. Unfortunately, it can only be queried, and not downloaded.

Out of the 19 279 recording from 17 speakers that the SWARA Speech Corpus holds, my application only uses only 18 306. The data from one user (having the speaker id TIM) has not been used altogether, due to an inconsistency I have found in the form of a mismatching between the number of audio files and the number of text transcriptions. Although the number should have been the same, I have discovered that there are 2 more audio files than there are text transcripts, which, in my opinion, rendered that particular data corrupt. So I decided not to use it at all. *Figure 29* displays the content of the final dataset.

No.	Speaker ID	Sex	Duration	No. of utts
1	BAS	F	1h 34' 30"	1493
2	CAU	F	1h 11' 35"	996
3	DCS	F	1h 50' 01"	1493
4	DDM	F	1h 09' 18"	996
5	EME	F	1h 53' 36"	1493
6	FDS	M	0h 57' 21"	996
7	HTM	F	1h 06' 27"	981
8	IPS	M	0h 58' 08"	996
9	PCS	M	1h 08' 03"	996
10	PMM	F	1h 01' 52"	921
11	PSS	M	1h 27' 45"	1486
12	RMS	M	1h 08' 56"	996
13	SAM	F	1h 43' 31"	1493
14	SDS	M	1h 01' 28"	996
15	SGS	M	0h 55' 22"	996
16	TIM	F	1h 09' 27"	975
17	TSS	M	1h 01' 54"	996

Figure 29. Dataset [56]

When it comes to the division of the data into *training data* and *test data*, I have decided to use three quarters of the data for the purpose of training and the remaining quarter for the testing.

I have stored all the needed data in 4 text files: train_mfccs, test_mfccs, train_transcripts and test_transcripts. I have processed all .wav audio files and their corresponding mel frequency cepstrum coefficients and stored them into train_mfccs and test_mfccs. These two files are not really humanly readable. The order of the entries in both the mfcc and transcript files for the train data and test data is the same. So, the first set of test mfccs corresponds to the first text entry (i.e. the first line) in the test_transcripts file.

The average duration of an audio file is approximately 4 seconds (3.9634227 seconds). Since every audio file is split into 25 millisecond frames, a recording has, on average, 160 frames. From one frame we derive 13 mel frequency cepstral coefficients, so, on average, a recording yields 2080 mel frequency cepstral coefficients.

As far as I could find through my research, there has only been one other paper on Romanian language speech recognition which made use of this dataset. That is the very recent May 2019 paper “Towards a Deep Speech Model for Romanian Language” [57]. The authors propose an end-to-end speech recognition system, based on the architecture of the acclaimed DeepSpeech. The network it proposes has 5 layers, of which the third one uses a bidirectional recurrent neural network and the last one is a fully connected layer. ReLU cells are being used for building the RNN model. The

ideal number of epochs they used for training the network was between 15 and 20. On top of the network, they experimented with adding different language models. With the default DeepSpeech language model, they obtained a word error rate around 60%. By using a Romanian language model, they managed to obtain a WER of 39%. When it comes to the CER, they managed to obtained a percentage as little as 15. Throughout the whole training process, they kept a batch size of 24.

3.4. Implementation Details

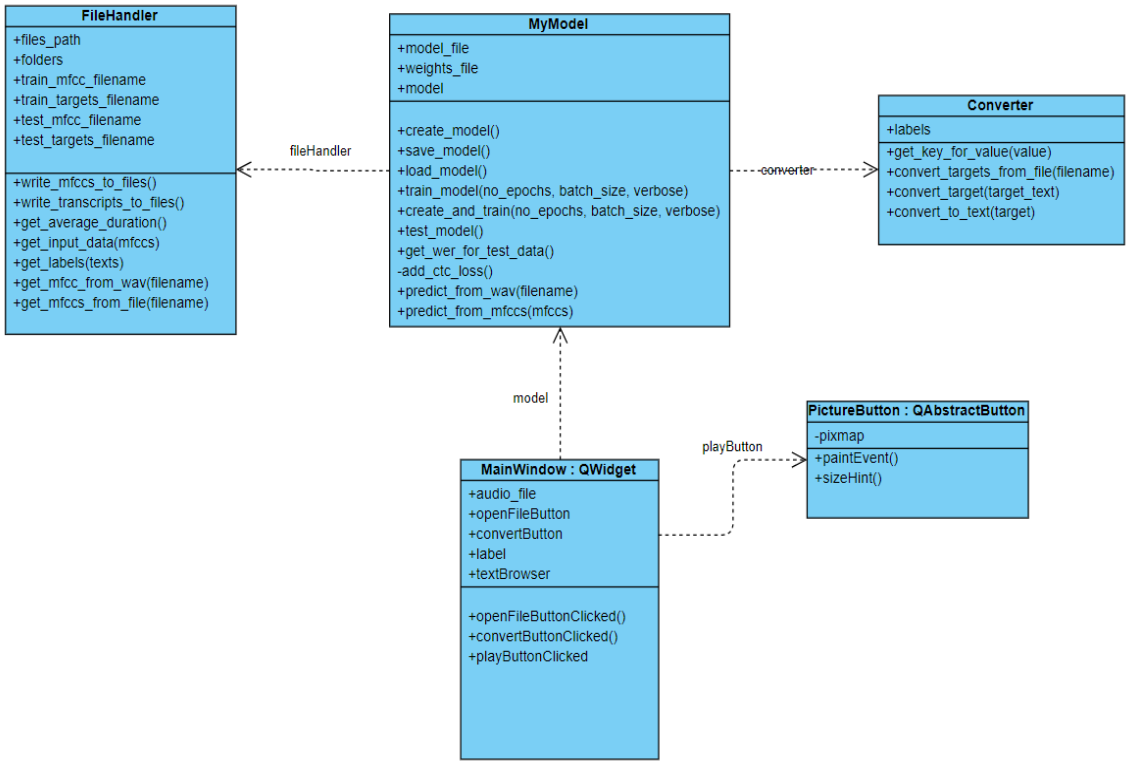


Figure 30. Detailed Class Diagram

Figure 30 holds the full class diagram of my application and I will keep referring to it throughout this subchapter, both in the back end, and in the front end part.

3.4.1. Back End

Since the main purpose of developing this application was building a neural network model suited for the task of transcribing speech to text, one should start off this subchapter by discussing how the creation of the model was achieved.

To begin with, the MyModel class was intended to be a wrapper over the Keras' Model class. As can be seen in the diagram, it provides methods for creating a model, for training it, testing it and saving it on disk as well as for loading a pre-existing model from a file on disk. All these methods are not meant to modify the pre-existent Keras Model API, but to make use of it and build on top of it in order to create a model that can recognize speech. It is to be noted that the last version of

the model is stored (centralized) in the *model* field of the MyModel class and all the methods make use of it.

The piece of code below represents the *create_model* method of the MyModel class. I have experimented with multiple neural network architecture on my journey of creating as accurate as possible a speech recognizer. The one here contains 5 long short-term memory layers, each of them having 33 neurons (the number of possible labels, including the blank label), followed by a *BatchNormalization* and a *TimeDistributed* layer, which are to be seen as utility layers. After all the layers are declared, the last layer is given a softmax activation function, with the purpose of computing a probability distribution over all 33 labels. Subsequently, the model is created, by initializing a Keras Model by giving it the input and the output layer. Finally, the model field of the class is updated to the newly created model and its output length is set as an identity lambda function. This property is to be used later.

```
def create_model(self):
    output_dim = len(self.converter.LABELS) + 1
    inputs = Input(name='input_data', shape=(None, 13))

    lstm1 = LSTM(output_dim=output_dim, return_sequences=True, implementation=2,
name='lstm1')(inputs)
    lstm2 = LSTM(output_dim=output_dim, return_sequences=True, implementation=2,
name='lstm2')(lstm1)
    lstm3 = LSTM(output_dim=output_dim, return_sequences=True, implementation=2,
name='lstm3')(lstm2)
    lstm4 = LSTM(output_dim=output_dim, return_sequences=True, implementation=2,
name='lstm4')(lstm3)
    lstm5 = LSTM(output_dim=output_dim, return_sequences=True, implementation=2,
name='lstm5')(lstm4)

    batch_normalization = BatchNormalization(name='batch_normalization')(lstm5)
    time_dense = TimeDistributed(Dense(output_dim))(batch_normalization)

    prediction_layer = Activation('softmax', name='softmax')(time_dense)

    self.model = Model(inputs=inputs, outputs=prediction_layer)
    self.model.output_length = lambda x: x
```

Having created the model, one has to train it. As can be seen in the following code, in order to do so, one must first get the list of mel frequency cepstral coefficients that serve as a representation for the audio in the training set. These are the inputs of the neural network. Secondly, since this is an instance of supervised training, the actual transcription of the audio files must be provided as well. Although the transcriptions are saved as strings in a plain text file, my model only knows integers, and not letters. Because of this, the *convert_targets_from_file* method of the Converter class takes all the targets in a given file and converts a list of strings into a list of lists of integers. This way, each transcription is converted into a list of integers, where each integer corresponds to a letter (or a space), according to the mapping defined below in the *LABELS* dictionary.

```
def train_model(self, no_epochs, batch_size, verbose):
    mfccs = FileHandler.get_mfccs_from_file(self.fileHandler.train_mfcc_filename)
    train_mfccs = FileHandler.get_input_data(mfccs)
    targets = self.converter.convert targets from file(self.fileHandler.train targets filename)
    train transcriptions= FileHandler.get labels(targets)

    train_input = {**train_mfccs, **train_transcriptions}
    train_output = {'ctc': train_mfccs['ctc']}

    self.model.fit(epochs=no epochs, x=train input, y=train output, batch size=batch size,
           verbose=verbose)
```

```
LABELS = {'<space>': 0, 'a': 1, 'b': 2, 'c': 3,'d': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8,
          'i':9, 'j': 10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16,
          'q': 17, 'r': 18,'s':19,'t': 20,'u': 21, 'v': 22, 'w': 23, 'x': 24,
          'y': 25, 'z': 26, 'u\u0103': 27, 'u\u00e2': 28, 'u\u00ee': 29, 'u\u0219': 30, 'u\u021b': 31}
```

However, in between creating the model and training it two extra *Lambda* layers and some further *Input* layers are added, in order to integrate the CTC algorithm. This method was inspired from the GitHub user lucko515's speech recognition solution [58]. The input layers it adds are straight-forward: one for the labeled transcripts, one for the actual length of the input (because the final input will be padded and one needs to retain its real length) and one for the length of the labeled transcription, for the same reason. All these *Input* layers are defined here and not in the *create_model* method, where it might be more natural to define them, due to the fact that they are used when creating the final two *Lambda* layers. The *output_length* layer just applies the identity function on the input length, signaling that the length of the output will be equal to that of the input. The most interesting layer is the *loss_layer*, which is also the output layer, which is but a custom lambda function, *ctc_lambda_function*, which just calls *K.ctc_batch_cost* with the four given arguments. This method calculates the CTC loss, also referred to as the value of the CTC cost function, which was mentioned previously, when discussing Connectionist Temporal Classification.

In the end, a new model is created, based on the old one, but with an enhanced input (3 extra layers) and a different output. Now instead of outputting the probability distribution over the possible labels, it gives the value of the CTC cost function. This new model will be used for training and testing, but not for predicting.

```
def add_ctc_loss(self):
    labeled_transcripts = Input(name='labeled_transcripts', shape=(None,), dtype='float32')
    input_length = Input(name='input_length', shape=(1,), dtype='int64')
    label_length = Input(name='label_length', shape=(1,), dtype='int64')
    output_length = Lambda(self.model.output_length)(input_length)
    loss_layer = Lambda(self.ctc_lambda_function, output_shape=(1,), name='ctc')(
        [self.model.output, label_data, output_lengths, label_lengths])

    self.model = Model(
        inputs=[self.model.input, labeled transcripts, input length, label length],
        outputs=loss_layer)
```

There is nothing particularly notable about testing the model, since the method responsible for it only gathers the testing data in the exact same manner the training data was gather, and then delegates the responsibility to the *evaluate* method. Interestingly enough, making predictions proved to be the more eventful part.

The challenge I faced when implementing the *predict_from_mfccs* method was that I was dealing with a model created for outputting a loss value, and not a list of probabilities over labels. I somehow had to revert to the initial model, the one which gave a softmax output, and not just a float value. The way I dealt with this situation is I took the input of the *input_data* layer and the output of a *softmax* layer and I have created a new model out of these, whose weights I set to be those

of the already trained full model. These are the lines of code corresponding to my solution:

```
self.model = Model(inputs=self.model.get_layer('input_data').input,
        outputs=self.model.get_layer('softmax').output)

self.model.load weights(self.weights file)
```

Having changed the model in such a way that it is suited to make predictions, there is nothing left but calling the built-in *predict* method of the Keras model. However, as mentioned before, this only returns a probability distribution for each frame in the audio. On top of this, the CTC algorithm must be applied, in order to make sense of the probabilities and give the most likely transcription. This is being achieved by calling *K.ctc_decode*, a method belonging to the TensorFlow back end of Keras. By means of a dynamic programming algorithm, it manages to carry out the prediction process to an optimal end.

```
decoded = K.ctc_decode(prediction, output_length)
predicted ints = (K.eval(decoded[0][0]) + 1).flatten().tolist()
predicted text = self.converter.convert to text(predicted ints)
```

3.4.2. Front End

Having seen how the neural network is created, trained, tested and how it makes predictions, let us move on to the part of the project which makes it accessible to the end-user: the UI. Going back to the class diagram, the two classes which deal with this aspect are *MainWindow* and *PictureButton*. The latter is a lightweight class which implements Qt's *QAbstractButton* and is just a custom widget which permits transforming an image into a button (in my case, the image of the play symbol). The former holds the two basic buttons (the one which opens the wav file and the one that converts the speech into text), the previously mentioned play button, the label attached to the text box where the transcript is to be displayed and the text box itself. The class also holds handler for when all the above buttons, along with one another important field: a model of type *MyModel.*
MainWindow also holds an *audio_file* field, which is set whenever the user chooses an audio file from disk, by clicking the "Open .wav file" button. The widget which enables the user to make a choice is the *QFileDialog*, as seen below.

```
def openFileButtonClicked(self):
    print("Open file button clicked")
    name = QFileDialog.getOpenFileName(self, 'Open File')[0]
    if name == '':
        return
    self.audio_file = name
    self.convertButton.setEnabled(True)
    self.playButton.setEnabled(True)
```

Naturally, the *model* field plays a key role in the handler of the button that does the speech to text conversion. Thus, whenever the conversion button is clicked, the *predict_from_wav* method is called on the model, with the name of the chosen audio file given as a parameter.

```
def convertButtonClicked(self):
    print("Convert button clicked")

    prediction = self.model.predict_from_wav(self.audio_file)
    print("Prediction is: ", prediction)
    self.textBrowser.clear()
    self.textBrowser.setText(prediction)
```

The way all these handler methods are linked to the buttons is through the following line of code:

```
self.widgetName.clicked.connect(self.handlerMethodName)
```

3.5. Outcomes

While working on this project, I have tried multiple neural network architectures, with the aim of finding the one that performs the best. In the process, although I have not managed to obtain a STT system that is accurate enough to be usable, I have derived some insightful observations regarding performance and training patterns.

There were 2 hyperparameters (a and b) and 2 parameters (c and d) I kept adjusting in hopes of getting a better performing network:

a) the number of epochs
b) the batch size
c) the number of layers
d) the number of neurons in the layers

When deciding the number of epochs over which to train the data, there is a thin line between training enough and training for too long (overfitting). For 4 or less LSTM layers with varying amounts of neurons, I have come to the same conclusion: for any number of epochs larger than 15, after the 15^{th} epoch the progress tends to stall and the loss will not decrease. Yet, for 5 layers, training the model for more than 15 epochs does make sense.

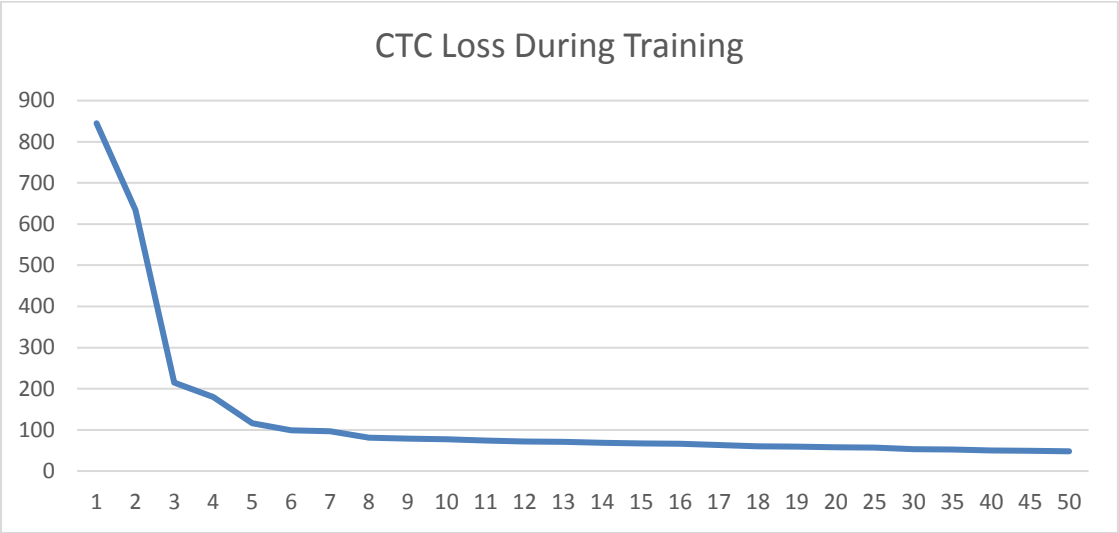
For the batch size, I have found 20-25 to yield the best results. I have also found the batch size to be a sensitive hyperparameter es1pecially when it comes to smaller datasets (like the one at hand).

Regarding the number of layers, the models having 5 LSTM layers performed better than those having 4 LSTM layers, and those having 4 LSTM layers performed better than those having only 3.

All in all, the minimum loss (48) was experienced with a model having the following characteristics and being trained under the following conditions:

- 5 LSTM layers, of 33 neurons each
- batch size of 20
- 50 epochs

The total training of this model took 20 hours and this is how the training progress looks like throughout the epochs:



A trend I have discovered that occurs during the training process is that is starts at a loss around 1000 and then quickly decreases for the first few epochs. When it reaches 90 however, the decrease happens very slowly; it could get stuck around 90 for 3 or 4 epochs on end.

Without the last layers, the *BatchNormalization* and *TimeDistributed*, the training registers a loss around 600 for hours on end− at least 9, according to my tests.

Conclusions

This thesis introduced the problem of speech recognition. Furthermore, it introduced the problem of achieving speech recognition by means of using long short-term memory neural networks, an improved type of recurrent neural networks. It also presented a speech-to-text end to end application, built using the Keras framework, with a TensorFlow back end, along with the observations that were drawn from programming it. The main functionalities of Keras framework were also presented in the thesis.

As regards future work, an interesting addition would be to incorporate a language model. One could consider the model proposed by the *Towards a Deep Speech Model for Romanian Language* paper. Also, when bigger Romanian speech corpora will emerge, it would be really interesting to train the most performant neural network architecture obtained in this thesis on them and see how much better the same architecture performs when trained on a substantially larger dataset.

One might also try replacing the long short-term memory units with gated recurrent units (GRUs). Since there are a lot of successful papers solving speech recognition to convolutional neural networks, a switch to a convolutional neural network might as well be attempted, although that clearly exceeds the scope of this thesis.

Bibliography

[1] C. Olson. [Online]. Available: https://www.campaignlive.co.uk/article/just-say-it-future-search-voice-personal-digital-assistants/1392459. [Accessed August 2019].

[2] https://medium.com/@SheLovesTechOrg/voiceitt-the-israeli-startup-making-voice-recognition-accessible-to-all-210025b0eb79, 2019. [Online]. [Accessed August 2019].

[3] T. Sainath, R. Weiss, S. Andrew, K. Wilson and O. Vinyals, "Learning the speech front-end with raw waveform CLDNNs," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[4] M. Ravanelli and Y. Bengio, "Speaker recognition from raw waveform with SincNet," in *2018 IEEE Spoken Language Technology Workshop (SLT)*, 2018.

[5] Grand View Research, "Voice and Speech Recognition Market Size, Share & Trends Analysis Report, By Function, By Technology (AI, Non-AI), By Vertical (Healthcare, BFSI, Automotive), And Segment Forecasts, 2018 - 2025," 2018.

[6] J. Kincaid, "https://medium.com/descript/a-brief-history-of-asr-automatic-speech-recognition-b8f338d4c0e5," [Online]. [Accessed 08 May 2019].

[7] C. Boyd, "https://medium.com/swlh/the-past-present-and-future-of-speech-recognition-technology-cf13c179aaf," [Online]. [Accessed 09 May 2019].

[8] X. Huang, J. Baker and R. Reddy, "Huang, Xuedong, James Baker, and Raj Reddy. "A historical perspective of speech recognition.," *Commun. ACM 57.1* , pp. 94-103, 2014.

[9] A. Graves, S. Fernandez, F. Gomez and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, 2006.

[10] "http://what-when-how.com/video-search-engines/speech-recognition-audio-processing-video-search-engines/," [Online]. [Accessed 22 August 2019].

[11] J. Garofolo, "TIMIT acoustic phonetic continuous speech corpus," *Linguistic Data Consortium*, 1993.

[12] V. Panayotov, G. Chen, D. Povey and S. Khudanpur, "Librispeech: an ASR corpus based on public domain audio books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[13] J. Garofolo, "Wall street journal-based continuous speech recognition (csr) corpus," *Linguistic Data Consortium, Philadelphia*, 1994.

[14] A. Stan, F. Dinescu, C. Ţiple, Ş. Meza, B. Orza, M. Chirilă and M. Giurgiu, "The SWARA speech corpus: A large parallel Romanian read speech dataset," in *2017 International Conference on Speech Technology and Human-Computer Dialogue (SpeD)*, 2017.

[15] wer_are_we, "https://github.com/syhw/wer_are_we," [Online]. [Accessed 09 May 2019].

[16] C. Lüscher, E. Beck, K. Irie, M. Kitza, W. Michel, A. Zeyer, R. Schlüter and H. Ney, "RWTH ASR Systems for LibriSpeech: Hybrid vs Attention-w/o Data Augmentation," *arXiv preprint arXiv:1905.03072,* 2019.

[17] H. Hadian, H. Sameti, D. Povey and S. Khudanpur, "End-to-end Speech Recognition Using Lattice-free MMI," 2018.

[18] K. J. Han, A. Chandrashekaran, J. Kim and I. Lane, "The CAPIO 2017 conversational speech recognition system," *arXiv preprint arXiv:1801.00059,* 2017.

[19] "https://sites.google.com/site/textdigitisation/qualitymeasures/basics," [Online]. [Accessed 25 August 2019].

[20] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966.

[21] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satsheesh, S. Sengupta, A. Coates and A. Ng, "Deep Speech: Scaling up end-to-end speech recognition," *CoRR,* 2014.

[22] "https://distill.pub/2017/ctc/," Hannun, Awni, 2017. [Online]. [Accessed 22 August 2019].

[23] A. Hannun, "Sequence Modeling with CTC," *Distill,* 2017.

[24] M. Caudhill, "Neural Network Primer: Part I," *AI Expert,* pp. 46-52, 1987.

[25] J. Brownlee, "https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/," [Online]. [Accessed 10 June 2019].

[26] S. Srivastava, S. Anupam and T. Ritu, "Machine Translation: From Statistical to modern Deep-learning practices.," *arXiv preprint arXiv:1812.04238,* 2018.

[27] S. Barker, "https://shanebarker.com/blog/seo-practices-for-machine-learning/," 2019. [Online]. [Accessed 10 June 2019].

[28] N. Lathia, "https://towardsdatascience.com/deep-learning-medical-diagnosis-c04d35fc2830," [Online]. [Accessed 10 June 2019].

[29] S. Bhattacharjee, "https://towardsdatascience.com/predicting-professional-players-chess-moves-with-deep-learning-9de6e305109e," [Online]. [Accessed 10 June 2019].

[30] T. Capes et al, "Siri On-Device Deep Learning-Guided Unit Selection Text-to-Speech System," in *INTERSPEECH*, 2017.

[31] "https://www.researchgate.net/publication/321259051_Prediction_of_wind_pressure_coefficients_on_building_surfaces_using_Artificial_Neural_Networks/figures?lo=1," [Online]. [Accessed 03 May 2019].

[32] "https://explained.ai/matrix-calculus/index.html," [Online]. [Accessed 05 May 2019].

[33] P. A. Rutecki, "Neuronal excitability: voltage-dependent currents and synaptic transmission," *Journal of clinical neurophysiology: official publication of the American Electroencephalographic Society 9.2,* pp. 195-211, 1992.

[34] A. L. Hodgkin and A. F. Huxley, " A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology,* pp. 500-544, 1952.

[35] A. Ng, "https://www.coursera.org/learn/neural-networks-deep-learning," [Online]. [Accessed 04 May 2019].

[36] "https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6," [Online]. [Accessed 03 May 2019].

[37] "https://sebastianraschka.com/faq/docs/relu-derivative.html," [Online]. [Accessed 04 May 2019].

[38] A. Geitgey, "https://medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471," [Online]. [Accessed 04 May 2019].

[39] "https://hackernoon.com/gradient-descent-aynk-7cbe95a778da," [Online]. [Accessed 04 May 2019].

[40] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling,* p. 1, 1988.

[41] A. Karpathy, "http://karpathy.github.io/2015/05/21/rnn-effectiveness/," [Online]. [Accessed 05 May 2019].

[42] M. Venkatachalam, "https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce," [Online]. [Accessed 06 May 2019].

[43] Skymind, "https://skymind.ai/wiki/lstm#two," [Online]. [Accessed 06 May 2019].

[44] Z. C. Lipton, J. Berkowitz and C. Elkan, "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019,* 2015.

[45] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of*

the IEEE, pp. 1550-1560, 1990.

[46] M. Nielsen, "http://neuralnetworksanddeeplearning.com/chap5.html," [Online]. [Accessed 07 May 2019].

[47] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult.," *IEEE transactions on neural networks*, pp. 157-166, 1994.

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, pp. 1735-1780, 1997.

[49] C. Olah, "http://colah.github.io/posts/2015-08-Understanding-LSTMs/," [Online]. [Accessed 06 May 2019].

[50] F. A. Gers, J. Schmidhuber and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural computation*, pp. 2451-2471, 2000.

[51] W. Zaremba and I. Sutskever, "Learning to execute," *arXiv preprint*, 2014.

[52] K. Cho et al., "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[53] F. A. Gers and J. Schmidhuber, "Recurrent nets that time and count," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 2000.

[54] D. Trandabăț, E. Irimia, V. Mititelu, D. Cristea and D. Tufiș, The Romanian Language in the Digital Age, Springer, 2012.

[55] A. Stan, F. Dinescu, C. Tiple, S. Meza, B. Orza, M. Chirila and M. Giurgiu, "The SWARA Speech Corpus: A Large Parallel Romanian Read Speech Dataset," in *Proceedings of the 9th Conference on Speech Technology and Human-Computer Dialogue (SpeD)*, Bucharest, Romania, 2017.

[56] V. Mititelu, E. Irimia and D. Tufis, "CoRoLa—The Reference Corpus of Contemporary Romanian Language.," in *LREC*, 2014.

[57] "https://speech.utcluj.ro/swarasc/," [Online]. [Accessed 23 August 2019].

[58] M. Panaite, S. Ruseti, M. Dascalu and S. Trausan-Matu, "Towards a Deep Speech Model for Romanian Language," in *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*, 2019.

[59] lucko515, "https://github.com/lucko515/speech-recognition-neural-network," [Online]. [Accessed 26 August 2019].

[60] "https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network,"

[Online]. [Accessed 03 May 2019].

[61] "https://en.wikipedia.org/wiki/Heaviside_step_function," [Online]. [Accessed 03 May 2019].

[62] M. Nielsen, "http://neuralnetworksanddeeplearning.com/chap2.html," [Online]. [Accessed 05 May 2019].

[63] M. Nguyen, "https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21," [Online]. [Accessed 06 May 2019].

[64] "https://www.researchgate.net/publication/241519992_Characteristics_of_small_boat_acoustic_signatures/figures?lo=1," [Online]. [Accessed 09 May 2019].