FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Continuous Speech Recognition by Neural Networks
Student:	Adam Zvada
Supervisor:	Ing. Miroslav Skrbek, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2018/19

Instructions

Explore possible methods for continuous speech recognition using neural networks. Consider recurrent neural networks and the possibility of using neural Turing machines. Based on the investigation and in agreement with the supervisor, select the appropriate solution for the robot NAO. Benefit from existing libraries providing implementations of the required methods. Validate the proposed solution on real data. Specify the scope of the work, together with the supervisor.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 13, 2018

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Informatics

Bachelor's thesis

Continuous Speech Recognition by Neural Networks

Adam Zvada

Supervisor: Ing. Miroslav Skrbek Ph.D

May 15, 2018

Acknowledgements

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 15, 2018   . . . .

Citation of this thesis

Abstrakt

Cilem teto bakalarske prace je prozkoumat oblast rozpoznavani reci a neuronovych siti, s cile implemenotovat end-to-end rozpoznavani reci jako interface pro Robot NAO.

U V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova   neuronové sítě, rekurentní neuronové sítě, rozpoznávání řeči, TensorFlow, CTC, LSTM, Robot NAO, Python

Abstract

The aim of this bachelor thesis is to explore the field of speech recognition and neural networks with a goal to implement end-to-end speech recognizer as voice-user interface for Robot NAO.

The proposed speech recognizer architecture is consisted from three main components: feature extraction of speech signal using mel-frequency cepstrum coefficients method, recognizer as recurrent neural networks with long-short-term-memory cells, and connection temporal classification algorithm for the final transcription.

The result of this work is end-to-end speech recognizer trained on VCTK corpus and implemented in programming language Python, using deep learning library TensorFlow.

Contents

List of Figures

Introduction

The problem of speech recognition (SR) has been an important research topic since as early as the 70s. Recently, the field of SR has seen major advances because of the rise of computing power (GPUs) which allowed innovation in machine learning and artificial intelligence algorithms. Now we have access to voice control through speech recognition in mobile devices, computers, smart TVs or even fridges.

Before the emergence of deep learning, researchers often utilized other classification algorithms such as Hidden Markov Model (HMM) with many complex handcrafted components. The field is now gradually moving towards end-to-end speech recognizer using just a neural networks which learns to transcribe an audio sequence signal directly to a word sequence, one character at a time. Therefore, all the handcrafted components would be replaced with a just one learning model.

In this thesis, we present the concept of artificial neural networks (ANN), basics of the internal network architecture and explained the training phase of ANN. We extend the knowledge of neural networks by introducing recurrent neural networks and most importantly we cover how speech recognition system works and how can we build end-to-end SR using neural networks.

Our goal is to get theoretical overview in this field and implement end-to-end speech recognizer using neural networks and TensorFlow library which would be used in Robot NAO as voice-user interface on Robot NAO.

Neural Network

Neural networks have a remarkable ability to derive meaning from complicated data. They can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques[32]. Even though they have been around since the 1950s, it is only in the last decade when they started to outperform robust system or even humans in specify tasks. However, they require a huge amount of training examples and computational power to be trained for preforming a reasonable prediction. Fortunately, GPUs has seen enormous increase in performance[1] and 90% of the data in the world today has been created in the last two years alone, at 2.5 quintillion bytes of data a day[18]. That's why ANN is big topic in Computer Science and in the technology industry and it currently provides the best solutions to many problems such as speech recognition, image recognition, and natural language processing.

1.1 Inspiration in Nature

Artificial neural network (ANN) is heavily inspired by the way how biological neural networks process information in the human brain. Even though our brain is extremely complex and still not fully understand, we just need to know how information is being transferred. The basic building block is nerve cell called *neuron*. It receives, processes, and transmits information through electrical and chemical signals[27]. It's estimated that an average human has 86 billion neurons[9].

As shown on Figure 1.1, *dendrites* are extensions of a nerve cell that propagate the electrochemical stimulation received from other neurons to the cell body. You may think of them as inputs to neuron, whereas neuron's output is called *axon*, a long nerve fiber that conducts electrical impulses away from

---

[1]GPUs are explicitly designed to handle multiple matrix calculations at the same time. Evaluation and training of artificial neural networks are mostly matrix operations.
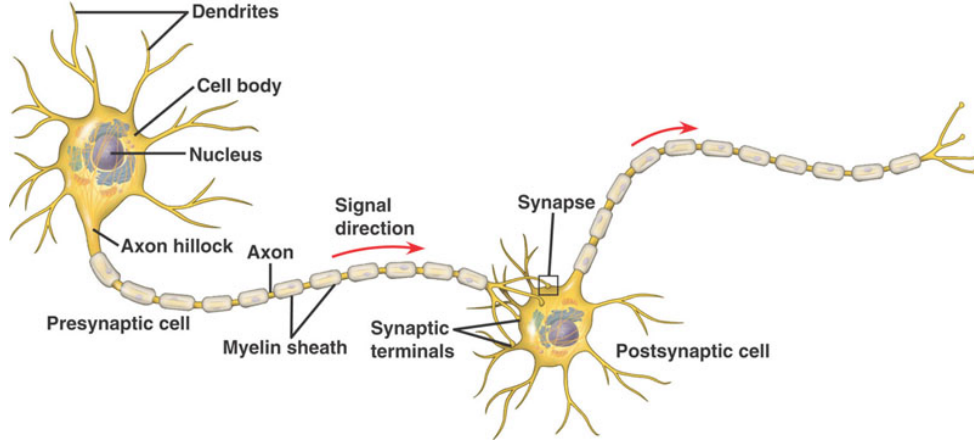
Figure 1.1: Illustration of nerve cell and communication flow

the cell body. The end of axon is branched to many axon terminals which can be again connected to other dendrites. The connection is managed by *synapses* that can permit the passing of electrical signal to cell body. Once the cell reaches a certain threshold, an action potential will fire, sending the electrical signal down the axon to other connected neurons.

1.2 Artificial Neuron

Artificial neuron is a generic computational unit, basic building block for artificial neural network (ANN). It's simplified version of the biological counterpart and we are able to map parts of biological neuron with the artificial one. It takes n inputs represented as a vector $x \in \mathbb{R}^n$ which correspond to dendrites. Generaly artificial neuron produces single output $y \in \mathbb{R}$ as biological neuron where we call it axon. Each neuron's input $i = 1, 2, \ldots, n$ has assigned weight (synapse) $w_1, w_2 \ldots w_n$, they refer to the connection strength between neurons. Weights and same as for synapse are the backbone of learning because in training phases, they keep changing to produce wanted output. Inside the artificial neuron, input vector with their weights are combined and run through an activation function producing some output y. This process is illustrated in Figure 1.2.

1.3 Perceptron

Perceptron is the simplest ANN with just one neuron and since we covered the basic intuition about artificial neuron we may proceed further and take a look at how output is actually calculated. The equation for a perceptron can
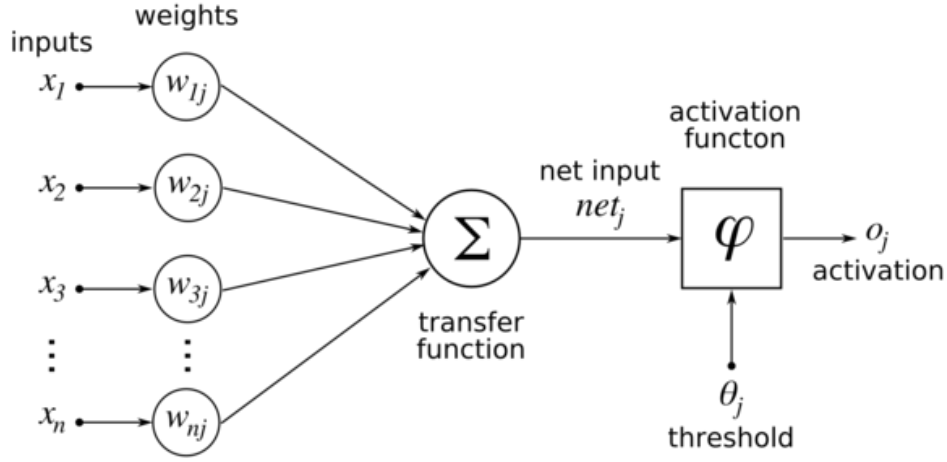
Figure 1.2: Illustration of nerve cell and communication flow

be written as

$$y = f(\sum_{i=1}^{N} w_i \cdot x_i + b) \qquad (1.1)$$

where

- x - input vector

- y - predicted output

- f - activation function

- w - weights

- b - bias

Perceptron is a basically linear classifier, therefore the data has to be linearly separable otherwise we would not be able to make the correct prediction. Problems such as speech recognition are not definitely linearly separable, however we can solve non-linear decisions for example by introducing another layer of neurons, thus creating *Multilayered Perceptron*.

1.3.1 Activation Functions

We have stated that biological neuron fires electrical signal to other connected neurons whenever it reaches a certain threshold of incoming electrical impulses. Activation function is based on that concept and inside an artificial neuron it is used for calculating output signal via equation 2.1. It introduces non-linear properties to our ANN and without an activation function would

be just a regular linear regression model. Nowadays many different activation function are being used and their performance varies from model to model.

List of some activation function:

- *Sigmoid*

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

- *Hyperbolic Tangent*

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

- *ReLU*

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

- *Softmax*

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}, \quad i = 1, 2 \ldots J$$

where i is number of output

1.3.2 Bias

We can think of bias as a value stored inside neuron and being used to calculate it's output. The bias value allows the activation function to be shifted to the left or right, to better fit the data.

1.4 Topology of Artificial Neuron Network

Basic ANN as feedforward model is a directed graph with nodes as neurons and edges with weights representing connection to other neurons. ANN can be divided to three important layers as shown in Figure 1.3. Yellow nodes is an input layer which takes input data, dimension of input vector has to correspond to number of input nodes. Hidden layer as the green nodes is most important to ANN and that is where the training and evaluation happens. Number of hidden layers and neurons needs to be in a good ratio between its size and its effectiveness. Output layer produces output vector as the prediction for given input.

1.4.1 Network Evaluation

ANN are sometimes called feedforward neural network. The reason behind is that the input is fed into the neuron and then forward to another layer, thus ANN are evaluated layer by layer. All neurons calculates the output using similar formula as Perceptron 2.1.
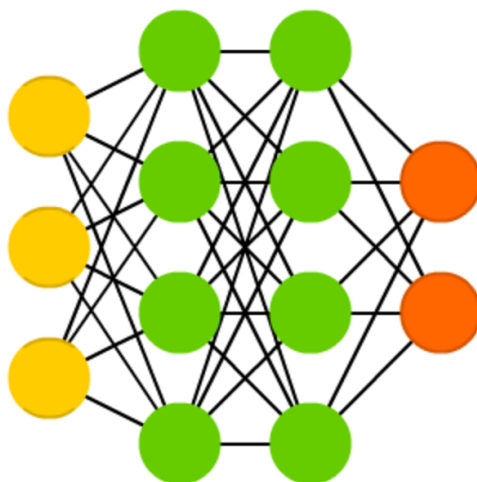
Figure 1.3: Basic topology of fully connected artificial neuron network with input vector of size 3, output vector of size 2 and two hidden layers.

1.5 Training

The greatest trait of ANN is ability to learn from given data and then make the best approximate prediction. The aim of the learning process is to find the most optimal values for network's weights and biases while minimizing error on predicated values. For ANN to learn we have to introduce training data consisted of input vector which will be fed to the network and desired output value (label) for calculating our loss. This approach is called supervised learning[2].

1.5.1 Loss Function

Loss function compares the prediction from ANN with the desired output and returns the error of the prediction. During a training ANN, the goal is to minimize given loss function. The most common and most intuitive loss function is Mean squared Error (MSE),

$$\text{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

1.5.2 Backpropagation

Backpropagation algorithm is responsible for the ability to learn from given training data. It is an iterative algorithm which for each training data from

---

[2]ANN can be also trained using unsupervised learning.

given training dataset backpropagates the error and adjust the weights and biases accordingly to get desired output.

1.5.2.1   Optimization

Backpropagation requires optimizer to minimize the error on the training data. We will describe backpropagation with using *gradient descent* as the most common optimization algorithm.

Weights and biases are updated using formula,

$$W_{jk}^l := W_{jk}^l - \alpha \frac{\partial E}{\partial W_{jk}^l} b_j^l := b_j^l - \alpha \frac{\partial E}{\partial b_j^l} \tag{1.2}$$

where $\boldsymbol{W_{jk}^l}$ is weight with connection between unit j in layer l and unit i in layer $l+1$, $\boldsymbol{b_j^l}$ is bias associated with unit i in layer $l+1$, $\boldsymbol{\alpha}$ is a learning rate [41], and $\frac{\partial E}{\partial W_{jk}^l}$ or $\frac{\partial E}{\partial b_j^l}$ can be interpreted as minimizing loss function with respect to given weight and bias respectively.

By applying a chain rule twice on the partial derivative of the loss function with respect to a weight, we get

$$\frac{\partial E}{\partial W_{jk}^l} = \frac{\partial E}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial W_{jk}^l} \tag{1.3}$$

where $\boldsymbol{z_j^l}$ is a sum of weighted inputs to unit j in layer l

$$z_j^l = b_j^l + \sum_{k=1}^{K} w_{jk}^l a_k^{l-1} \tag{1.4}$$

and $\boldsymbol{a_j^l}$ is an output of node j in layer l

$$a_j^l = f(z_j^l). \tag{1.5}$$

Let's calculate the last two products of equation 2.3:

$$\frac{\partial a_j^l}{\partial z_j^l} = f'(z_j^l) \frac{\partial z_j^l}{\partial W_{jk}^l} = \frac{\partial W_{jk}^l a_k^l}{\partial W_j k^l} = a_k^{l-1} \tag{1.6}$$

We introduce a new varibale $\boldsymbol{\delta_j^l}$ which represents the error in unit j in layer l and helps us to better understand and calculate real interested value of $\frac{\partial E}{\partial W_{jk}^l}$ and $\frac{\partial E}{\partial b_j^l}$.

$$\delta_j^l = \frac{\partial E}{\partial z_j^l} \tag{1.7}$$

We will simplify the error equation on neuron j in output layer L as

$$\delta_j^L = \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L}f'(z_j^L) \qquad (1.8)$$

Now we have enough information to reformulate equation 2.3 for output layer to

$$\frac{\partial E}{\partial W_{jk}^l} = \delta_k^L a_j^L. \qquad (1.9)$$

However, to be able to update weights inside the hidden layers, we have to redefine the calculation of δ_j^l. We know that the error produced by an output neuron is just influencing the output value but inside a hidden layer the produced error propagates to all following layers. Therefore we have calculate the δ_j^l where layer l is inside a hidden layer and take into account all δ^{l+1} from following layer $l+1$.

$$\delta_j^l = \frac{\partial E}{\partial z_j^l} = \sum_i \frac{\partial E}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial z_j^l} = \sum_i \frac{\partial E}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_i \delta_i^{l+1}W_{ij}^{l+1}f'(z_j^l)$$
$$(1.10)$$

where the sum index i iterates over all neurons in layer $l+1$ and Notice that we have substituted $\frac{\partial E}{\partial z_i^{l+1}}$ with δ_i^{l+1} which is calculated from previous iteration**??**. Finally, we may calculate all weights adjustments through the whole network as

$$W_{jk}^l := W_{jk}^l - \alpha\delta_k^l a_j^l \qquad (1.11)$$

where

$$\delta_k^l = \frac{\partial E}{\partial a_j^L}f'(z_j^L), \quad l = L \qquad (1.12)$$

or

$$\delta_k^l = \sum_i \delta_i^{l+1}W_{ij}^{l+1}f'(z_j^l), \quad l = 2,\dots,L-1. \qquad (1.13)$$

We won't be exampling the equation for biases adjustments because it follows a similar process shown above with just little changes, resulting to equation

$$b_j^l := b_j^l - \alpha\delta_l^j \qquad (1.14)$$

1.5.2.2   Backpropagation Algortihm

Backpropagation algortihm in pseudocode:

Algorithm 1 Backpropagation

1: Initialize network weights and biases
2: **for each** training data from training dataset **do**
3:     Forward pass and calculate network prediction for given training input
4:     Calculate error δ^L for output layer
5:     Calculate errors δ^l for hidden layers
6:     Update weights and biases using precalculated δ^l

Recurrent Neural Network

Neural networks are powerful learning models that achieve state-of-the-art results in a wide range of machine learning tasks. Nevertheless, they have limitations in the field of sequential data. Standard ANNs rely on the assumption of independence among the training examples but if data points are related in time or space then ANNs would not be the right model for the task[23].

Recurrent neural network (RNN) is type of neural network which is precisely designed to work with sequential data through time. The key difference is that RNN's neurons in hidden layer have a special edge (recurrent edge) to a next time step which can be interpreted as a loop. In RNN, the neuron's output is dependent on the previous computations which is sent through the recurrent edge. Basically, the recurrent edges or loops allow persistence of information from one time step to the next one as shown on Figure 2.1 [12].
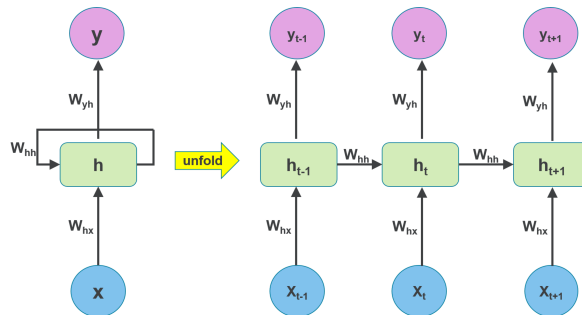
Figure 2.1: Simple RNN topology and illustration of unrolled RNN through time[39]

2.1 Evaluation

In 2.1 we may see simplification of evaluation process of RNN through the time steps. RNN's neuron cell in hidden layer takes two inputs, x_t and h_{t-1} which is value (hidden state) sent through the recurrent edge from previous time-step. The cell also produces two outputs, h_t as hidden state for upcoming time-step

$$h_t = f(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$

where f is arbitrary non-linear activation function, W_{hx} is matrix of conventional weights, W_{hh} is the matrix of recurrent weights and b_h is a bias. The second output from cell is y_t which outputs the predication using precalculated hidden state h_t,

$$y_t = W_{hy}h_t + b_y$$

where W_{hy} is matrix of output weights.

2.1.1 Softmax Fucntion

It is very common for RNN models to use *softmax* as activation function for output layer. Softmax function helps to get probability distribution of outputs so it's useful for finding most probable occurrence of output with respect to other outputs.

$$\text{softmax}(y)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \quad \text{for } j = 1, \ldots, K$$

Softmax is being used for calculating output value of y_t resulting to formula

$$y_t = \text{softmax}(W_{hy}h_t + b_y).$$

2.2 Training

Training a RNN is similar to training a traditional ANN. We also use the backpropagation algorithm, but since the parameters are shared by all time-steps in the network, the gradient at each output depends not only on the calculations of the current time-step, but also the previous time-steps[6].

2.2.1 Backpropagation Through Time

The most used algorithm to train RNN is *backpropagation through time* (BPTT), introduced by Werbos in 1990 [35]. BPTT is basically an extended version of backpropagation algorithm where we not only propagate the error to all following layers but also through the hidden states. We may think of it as unrolling the RNN to sequence of identical ANNs where the recurrent edge connects the sequences of neurons in hidden layer together as shown on Figure
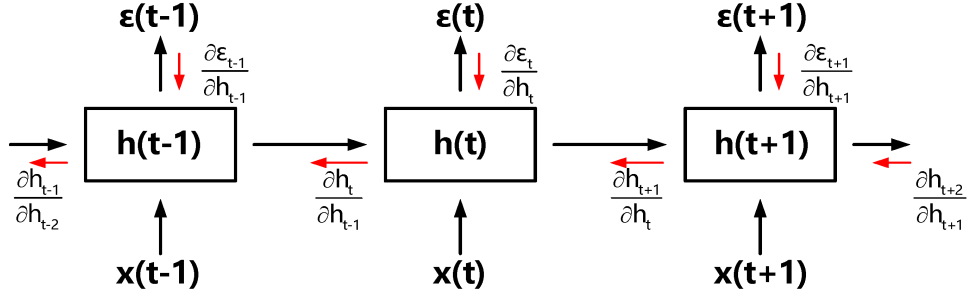
Figure 2.2: Deriving the gradients according to the backpropagation through time (BPTT) method. Notation for output value $\epsilon(t)$ corresponds to our y_t[24].

2.1 and 2.2. On Figure 3.2 2.2 is also indicated how the errors are propagated. The propagation of errors through hidden states allows the RNN to learn long term time dependencies. The calculated gradients of the loss function for defined parameter (W, b) through the sequence of unrolled RNN are then sum up, producing the final gradient for updating the weights or biases.

$$\frac{\partial E}{\partial W_i j^l} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W_i j^l}$$

where E is predefined loss function, W_{jk}^l is weight with connection between unit j in layer l and unit i in layer $l+1$, T is number of input sequences and $\frac{\partial E_t}{\partial W_{ij}^l}$ is calculated similarly as in backpropagation with just considering existence of recurrent edges

$$\frac{\partial E_t}{\partial W_{ij}^l} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}^l}$$

To compute the $\frac{\partial h_t}{\partial h_k}$ we use simple chain rule over all hidden states in interval $[k,t]$.

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

Putting equations together, we have the following relationship[28].

$$\frac{\partial E}{\partial W_{ij}^l} = \sum_{t=1}^{T} \sum_{j=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \left(\prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_{ij}^l}$$

2.2.2 Exploding and Vanishing Gradients

Even though, RNNs had achieved success in learning short-range dependencies, they haven't been showing any worth mentioning achievement with learn-

ing mid-range dependencies. That was mainly cause by problems of *vanishing* and *exploding gradients*, introduced in Bengio in 1994 [4].

The exploding gradient problem occurs when backpropagating the error across many time steps, that could lead to exponentially grow of gradient for long-term components. Basically, a small change in parameters at initial stages can get accumulated through the time-steps resulting to the exponentially grow. The values of weights can become so large as to overflow and result in *NaN* values.

The vanishing gradient problem refers to opposite behavior when the gradient values are shrinking exponentially fast and eventually vanishing completely. Gradient contributions from later time-steps become zero and the states at those steps doesn't contribute so we end up not learning long-range dependencies. Vanishing gradients aren't exclusive to RNNs, they also happen in deep ANN[7].

2.2.2.1   Solutions

To overcome problem with exploding gradient we can apply *gradient clipping* method. The values of the error gradient are checked against a predefined threshold value and clipped or set to that threshold value if the error gradient exceeds the threshold[1].   Another possibility is to use *ReLU* activation

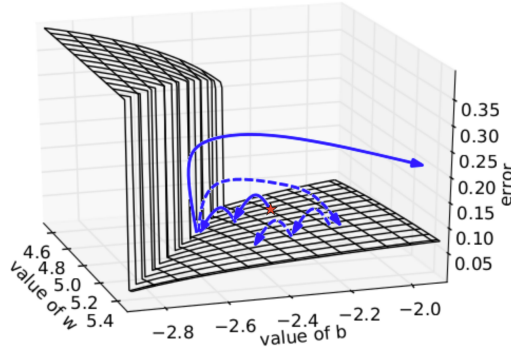Figure 2.3: Situation of using gradient clipping (dashed line) against the exploding gradient[28]

function which tends to reduce the the exploding gradient problem. To fix the problem of vanishing gradient is little more complicated. We can always try perform more careful initialization process but it does not always help. It requires different architecture approach achieved by updating the RNN neuron to more complex LSTM cells.

2.3 LSTM

Long-Short-Term-Memories (LSTM) is special kind of RNN cell, introduced by Hochreiter and Schmidhuber in 1997 [15]. Conventional RNNs are only just able to learn short-term dependencies because of vanishing gradient problem. However, LSTM does not get effected and it's capable of learning long-term dependencies.
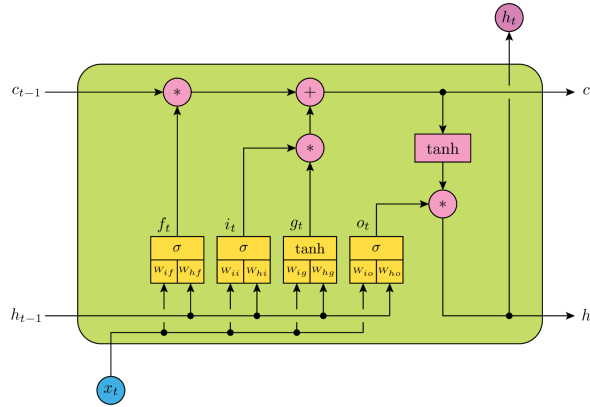
Figure 2.4: Diagram of LSTM cell[16].

As shown on Figure 2.4 we notice that LSTM is just more complex activation units. Similarly as basic RNN cell which propagates hidden state of h_t to another time-step and also as cell output, the LSTM cell has extra state denoted as c_t and called *cell state* and it's just being propagated to another time-step. The cell state is more of a cell's memory.

LSTM architecture follows stages during the evaluation where first we have to decide what information we want to get rid of from cell state, that is achieved applying formula using sigmoid function

$$f_t = \sigma(W_f h_{t-1} + W_f x_t + b_f) \tag{2.1}$$

and we call f_t as forget gate. Another step is to calculate so-called *input gate* denoted as i_t, it determines whether or not the input is worth preserving.

$$i_t = \sigma(W_i h_{t-1} + W_i x_t + b_i) \tag{2.2}$$

The third value is *memory gate* as g_t, it is using the input with the previous hidden state to observe the input in the context of the past.

$$g_t = \tanh(W_g h_{t-1} + W_g x_t + b_c) \tag{2.3}$$

Using equation 2.1, 2.2 and 2.3 we may calculate the new cell state using formula

$$c_t = f_t c_{t-1} + i_t g_t \tag{2.4}$$

15

Basically, c_t is constructed by applying the forget gate on the previous cell state and the memory gate gets augmented by the input gate. The last value to produce is hidden state which will be a sort of filtered cell state

$$h_t = \tanh(c_t)o_t \tag{2.5}$$

where o_t is called *output gate* and it augments input information using formula

$$o_t = \sigma(W_o h_{t-1} + W_o x_t + b_o) \tag{2.6}$$

The whole process of the LSTM cell evaluation is also illustrated on Figure 2.4.

2.4 Connectionist temporal classification

Connectionist temporal classification (CTC) is a loss function used for classification of sequential data, initially presented by Alex Graves in 2006 [14]. The idea of CTC is that the label is not generated directly by the RNN, but instead we calculate a probability distribution over all possible characters at every time-step.

For a sequence labelling task where the labels are from an alphabet L, we introduce extra unit as *blank* character, $\hat{L} = L \cup \{blank\}$. CTC consists of a softmax output layer which estimates the probabilities of observing the corresponding labels at particular times[13].

Let's denote that y^t_k of output unit k at time-step t is interpreted as the probability of observing label k at time t and input sequence x of length T. Now we can calculate a probability of path sentence $\pi \in \hat{L}$ using formula

$$p(\pi|x) = \prod_{t=1}^{T} y^t_{\pi_t}. \tag{2.7}$$

Now let's define many-to-one mapping β which simplifies the sentence path by striping the multiple trailing character to just one and then removing the *blank* characters altogether.

$$\beta(--hh--e--ll-lll-oo-) = \beta(-h-e-l-l-o-) = hello$$

We may calculate the marginal probability of the sequence l using the defined β mapping from given path:

$$p(l|x) = \sum_{\pi=\beta^{-1}(l)} (\pi|x) \tag{2.8}$$

This so-called *collapsing together* of different paths onto the same labelling is what allows CTC to use unsegmented data, because it removes the requirement of knowing where in the input sequence the labels occur. However, it

also makes CTC unusable for tasks where the location of the labels must be determined[13].

To decode the output for input sequence, we have to maximize the probability of sequence in respect to input data.

$$h(x) = \operatorname*{argmax}_{l} p(l|x) \tag{2.9}$$

For efficent calculation of $p(l|x)$ we use *backward-forward* algorithm with detail explanation on [14].

To use CTC for RNN training, we have to define the loss function for the BPTT algorithm. CTC loss function is derived from the principle of maximum likelihood with formula

$$E = -ln(\prod_{x,z} p(z|x)) = -\sum_{x,z} ln(p(z|x)) \tag{2.10}$$

where (x, z) are from the training dataset[13].

Speech Recognition

Speech recognition is the task of converting speech audio to text representation. It has been attracting researchers for many years with a goal to produce efficient speech recognizer, because it's a very easy and natural human-machine interface tool.

Speech recognition system takes audio signal as an input and predicts the text transcript. Arbitrary speech recognizers are normally divided into two important building blocks as shown on 3.1. The Feature Extractor block generates a sequence of feature vectors which are then fed to the recognizer block generating the correct output words.
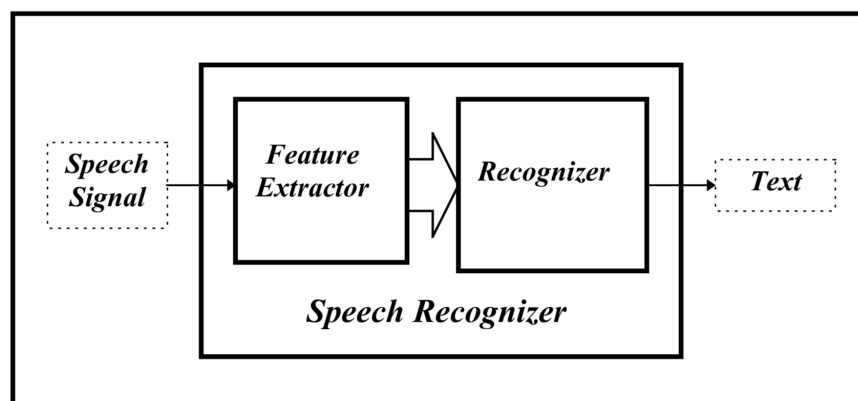
Figure 3.1: Basic building blocks of a Speech Recognizer [37]

3.1 Feature Extraction

The feature extraction (FE) block used in speech recognition should aim towards reducing the complexity of the problem, it should derive descriptive features from speech signal to enable a classification of sounds. It is needed because the raw speech signal contains other information besides the linguistic message which would be counterproductive for recognizer.

3.1.1 Preprocessing

It is advantageous to apply preprocessing to raw speech signal before moving to feature extraction block. Using some type of preprocessing leads to easier feature extraction and faster training phase.

Advantageous preprocessing method is to downsample given speech signal. Speech is mostly recorded with a sampling frequency of 44.1kHz or 48kHz, although speech signal has frequency components in the audio frequency from 20Hz to 20kHz[31]. That's because of Nyquist-Shannon sampling Theorem [19], a time-continuous signal that is band-limited to a certain finite frequency needs to be sampled with a double the maximum frequency. Since human speech has a relatively low bandwidth, mostly up to 8kHz. That means that sampling frequency of 16KHz is sufficient for speech recognition tasks[22].

Other part of preprocessing is to remove the parts between the recording starts and the user starts talking as well as after the end of speech. That helps to speed up the training phase because it reduces the size of training data.
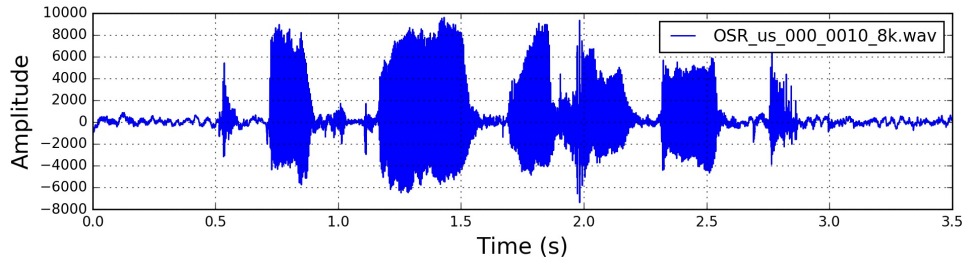
Figure 3.2: Illustration of raw speech signal from wav file with sampling frequency of 8kHz[11]

3.1.2 MFCC

Mel Frequency Cepstral Coefficents (MFCCs) are a feature widely used in speech recognition. They were introduced by Davis and Mermelstein in the 1980's, and have been state-of-the-art ever since [3].

MFCC mimics the logarithmic perception of loudness and pitch of human auditory system and tries to eliminate speaker dependent characteristics by excluding the fundamental frequency and their harmonics [21].
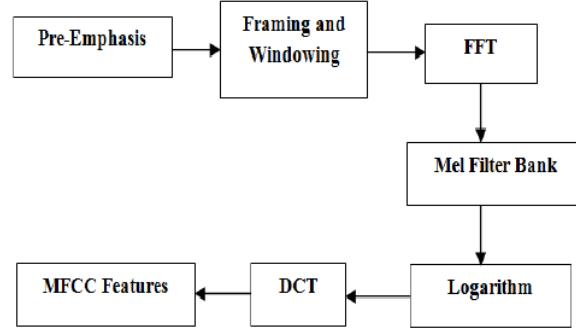
Figure 3.3: Steps of MFCC [30]

To obtain MFCC features we have to follow operation steps as shown on Figure 3.3:

- **Pre-Emphasis** - This step applies filter on the speech signal to amplify the high frequencies. It balances the frequency spectrum and avoids numerical problems during the Fourier transform operation.

$$y(t) = x(t) - \alpha x(t-1)$$

  where $x(t)$ is amplitude of signal in time t and α is filter coefficient which typical values are 0.95, $y(t)$ pre-emphasis speech signal.

- **Framing** - The process of segmenting the speech signal into a small frames with the length within the range of 10 to 40 milliseconds. Speech is non-stationary signal but we consider all frames behave stationary so they describe a phonemes. In SR we process overlapping frames because phonemes can dependent, resulting to smoother changes in values. Popular settings are 25 ms for the frame size, 10 ms stride (15 ms overlap) [11].

- **Windowing** - This step applies Hamming window function[33] on each speech signal frame. This is common operation for sound signal before applying FFT.

- **FFT** - This step converts all speech frames from time domain into frequency domain using Fast Fourier Transform (FFT) [34].

- **Mel Filter Banks** - This step applies the mel-filterbank which consists of triangular overlapping windows that are spread over the whole frequency range, outputting mel frequency spectrum. It mimics the non-linear human ear perception of sound, these filters are more discriminative at lower frequencies and less discriminative at higher frequencies [29].

21

- **_Logarithm_** - This step computes the logarithm of the mel frequency spectrum, to mimic the human perception of loudness because perceive loudness on a logarithmic scale.

- **_DCT_** - This step converts mel spectrum into time domain using Discrete Cosine Transform (DCT)[40], resulting to MFCC vectors.

We have just given a theoretical overview how MFCC is calculated, for more detailed explanation consider reading [21]. On Figure 3.4 is vector of MFCCs calculated from speech signal Figure 3.2 where number of cepstral coefficents is set to 13. We have extracted the features of speech signal and vectors of MFCCs can be fed to recognizer.
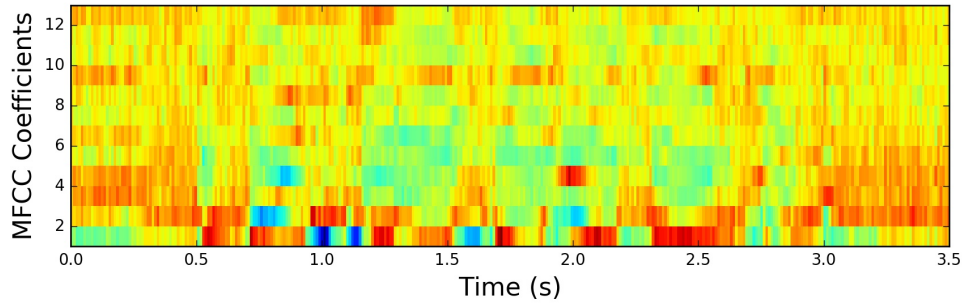
Figure 3.4: Vector of Mel Frequency Cepstral Coefficents through time.

3.2 Traditional Speech Recognizers

Historically, most speech recognition systems have been based on a set of statistical models representing the various sounds of the language to be recognized. We can define a problem of speech recognition as maximizing a probability of the word sequence given some utterance.

$$W^* = \underset{W}{\operatorname{argmax}} P(W|X)$$

where X are acoustic vectors and transcribed W^* word sequence. However, calculating directly W^* is a very difficult task. We may simplify it by using Bayes rule resulting to equivalent equation

$$W^* = \underset{W}{\operatorname{argmax}} P(X|W)P(W)$$

where the likelihood $P(X|W)$ is called the acoustic model and the prior $P(W)$ is the language model. In traditional speech recognizers we don't form words directly but we concatenating phonemes which are basic building block
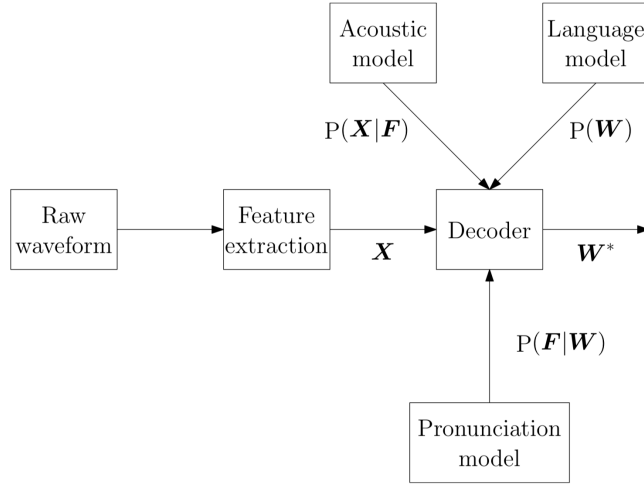
Figure 3.5: Diagram of traditional speech recognizer

of words and they are defined by pronunciation model. As shown on Figure 3.5, the decoder block works with language, acoustic and pronunciation model. The language model has a word sequences probabilities, while the acoustic model is generated by Hidden Markov Model (HMM) which is a tool for representing probability distribution over sequences of phonemes using pronunciation model[20].

In this thesis we just provide a basic overview how traditional speech recognizers, our main focus is on end-to-end recognizers.

3.3 End-to-End Speech Recognizers

Recent advances in algorithms and computer hardware have made it possible to train neural networks in an end-to-end fashion for tasks that previously required significant human expertise. All of the state-of-the-art speech recognizers were HMM-based, they required pronunciation, acoustic and language model which were hand-engineered and trained separately. Not only speech recognizers based on neural networks networks require less human effort than traditional approaches, they generally deliver superior performance [38]. Training independent components is complex and suboptimal compared to training all components as one. Because it replaces entire pipelines of hand-engineered components with neural networks, end-to-end learning allows us to handle a diverse variety of speech including noisy environments, accents and different languages[2]. End-to-end speech recognizers simplifies the training and deployment process altogether.

3.3.1 Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) were introduced in 2.4 a it is the best fit for end-to-end speech recognizer. Diagram on Figure 3.6 shows overview of the model architecture. Speech can be interpreted as time sequence, thus RNN with LSTM cells will be used since they are they are designed to deal with sequenctional data through time. The CTC make it possible to train RNNs for sequence labelling problems where the input-output alignment is unknown.
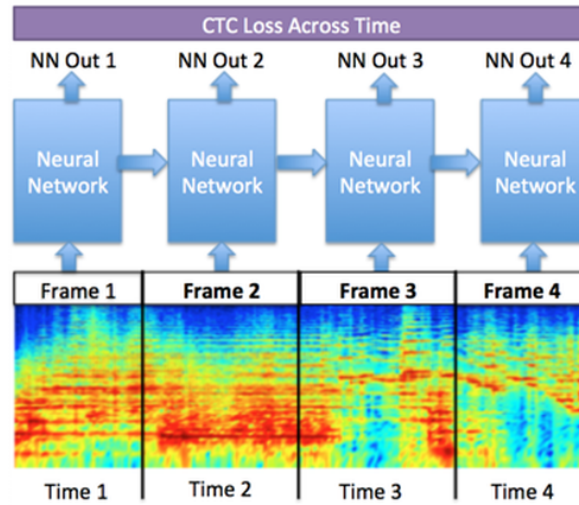
Figure 3.6: End-to-end speech recognizer diagram using CTC [26]

For example, this type of speech recognition model is used in Google Voice Search on Android and iOS [25].

3.3.2 Listen, Attend and Spell

Listen, Attend and Spell (LAS) is current state-of-the-art end-to-end speech recognizer [10].

It is based on *Attention Mechanism* [5], created from *Encoder* that reads reads and encodes a source sentence into a fixed-length vector and a *Decoder* that outputs a translation from the encoded vector. Attention Mechanisms are now considered one of the most exciting advancements in the field of AI.

LAS is consisted from an encoder recurrent neural network (RNN), which is named the *listener*, and a decoder RNN, which is named the *speller*. The listener is a pyramidal RNN that converts low level speech signals into higher level features. The speller is an RNN that converts these higher level features into output utterances by specifying a probability distribution over sequences of characters using the attention mechanism [8]. Both the listener and speller are trained jointly which is the motivation of end-to-end speech recognizers.

Implementation

The goal is to implement end-to-end speech recognizer using neural network. High-level concept, how the implemented speech recognition system works is illustrated on Figure 4.1. It takes a wav file as an input generated from given microphone and performs preprocessing and feature extraction. The data are fed to the recognizer which outputs the prediction of transcribed text from speech.
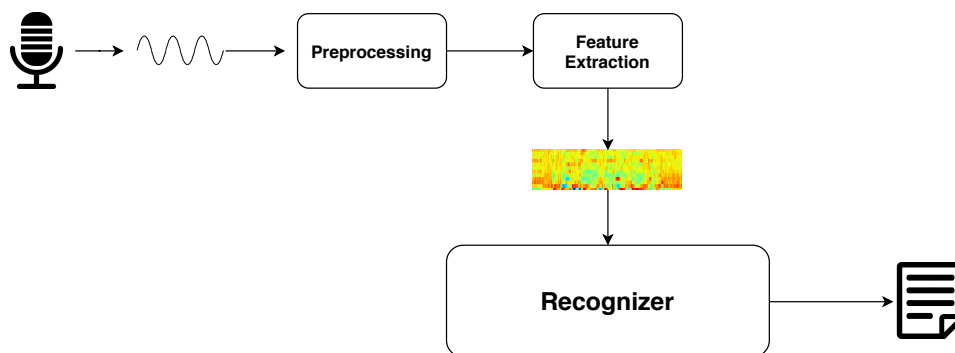
Figure 4.1: Speech Recognition System

The implemented recognizer is build on recurrent neural networks, therefore they need to be trained, in order to make a successful predictions. On Figure 4.2 is shown how the recognizer is being trained. It's done by providing speech and transcribed text from the training dataset. RNN feed-forwards all the vectors of MFCC and the RNN's output are processed by CTC. Obtaining the prediction text of the speech signal. Using backpropagation through time algorithm we update the weights and biases of RNN which minimize the error of the loss function resulting to better prediction in future.
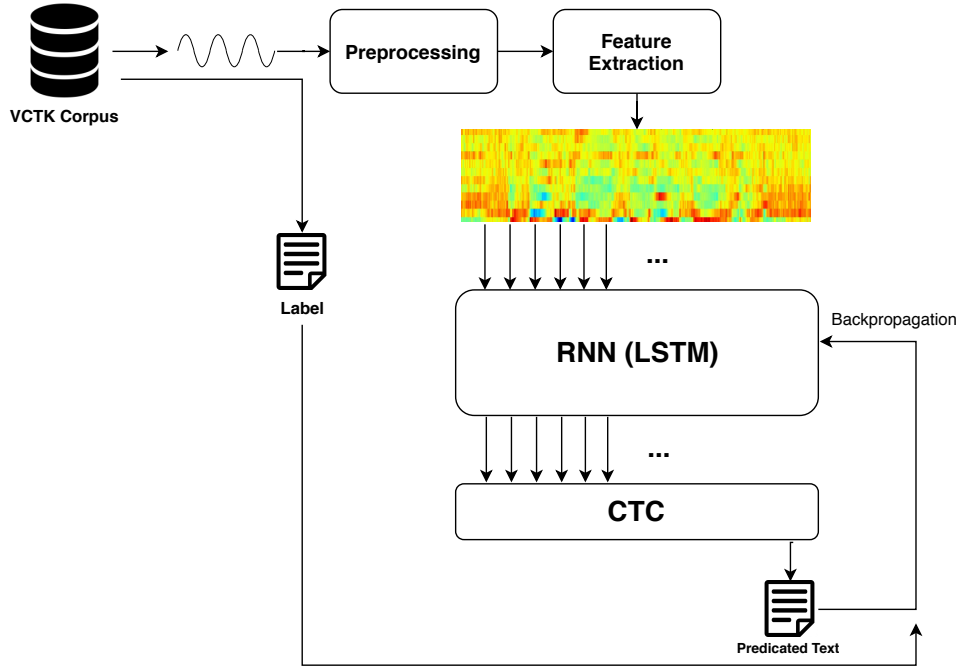
Figure 4.2: Diagram of the learning phase for the speech recognition system

4.1 Tools

4.1.1 Python

Speech recognition system is implemented in programming language `Python` which is currently most popular approach in machine learning and AI. Python is a very powerful, flexible, open source language that is easy to learn. The greatest strength however is wide range of libraries and frameworks for ML and AI.

4.1.2 TensorFlow

TensorFlow is open-source library developed by Google for deep learning and other algorithms involving large number of mathematical operations. The primary unit in TensorFlow is a tensor. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. These massive numbers of large arrays are the reason that GPUs and other processors designed to do floating point mathematics excel at speeding up these algorithms.

TensorFlow programs are structured into a construction phase that assembles a computational graph, and an execution phase that uses a session to execute operation in the graph. However, TensorFlow programs are hard to debug because of the structure. Fortunately, TensorFlow offers a built-in

function for visualization of the computation called TensorBoard.

4.2 Training Data

Training data are essential for neural networks performance and its quality, variety, and quantity determine the success of the learning models. Since we use approach of supervised learning for our recognizer, we have to provide labeled data.

4.2.1 Datset Base Class

In source code of the speech recognition system we have class `DatsetBase` which stores path to audios and transactions (labels) from our training dataset. It has also method `next_batch` which takes as a parameter `batch_size` and returns next batch of MFFC vectors and its labels. In method `next_batch` we retrieve speech signal data from audio file and perform preprocessing and feature extraction, then also the text labels are loaded from its file path. Upon the text labels is called preprocessing method which simplifies the text and eliminates all the non-alphabetic characters.

However, retrieving data from file system and performing processing and feature extraction upon them during a training phase is slowing down the process. One of the solution could be to prepare the data beforehand and store it as some variable which would lead to lower retrieving latency.

4.2.2 Numbers

Before using my learning model on large training dataset, I had been debugging and validating it on smaller dataset. I have used `Free Spoken Digit Dataset` from Github [17]. The dataset provides three english speakers with 1500 recordings, 50 recordings for each digit per speaker.

In source code we have a class `DigitDataset` which extends the base class `DatsetBase`. Class `DigitDataset` provides method called `read_digit_dataset` which takes argument of digit dataset path and stores all training data paths in audios and labels variables. They are later used in `next_batch` method.

4.2.3 VCTK Corpus

VCTK Corpus is training dataset which includes speech data uttered by 109 native speakers of English with various accents. Each speaker reads out about 400 sentences, most of which were selected from The Herald newspaper [36]. Even though, this dataset was designed to maximize the contextual and phonetic coverage for HMM-based speech recognizers, we might as well use it for ANN-based speech recognizer. The dataset size is around 15GB which is

still not enough to create robust production ready speech recognizer but it's enough for the purpose of this thesis.

4.3 Config Reader

To efficiently use different hyperparameters, datasets or feature extraction configurations. We run the speech recognizer training with *YAML* configuration file. Class `ConfigReader` handles parsing of the *YAML* file and as object it is passed to the main training method, where TensorFlow computational graphs is being constructed using the provided parameters from the object.

4.4 Preprocessing and Feature Extraction

4.4.1 Audio

In source code we have python file `audio_utils` with implements a function called `audiofile_to_input_vector`. The function takes as parameter file path to wav file and the number of cepstrum coefficients. First it loads the wav file from the file system and downsize the sample rate to 16kHz as a part of preprocessing. Even this reduced sample rate contains enough speech information for our recognizer to make successful predication. Then feature extraction is called upon the preprocessed wav file which is done by MFCC. Library `python_speech_features` provides implementation of MFCC method, we just need to configure the used parameters such as the number of cepstrum coefficients, length of window or the length of overlap.

4.4.2 Text

In python file `text_utils` we have function `get_refactored_transcript` which takes string and performs multiple operations for simplification. It converts string to lowercases, eliminates all non-alphabetic characters besides the spaces between words. Then string is converted to *numpy* array of characters which gets encoded to integers values. Thanks to the encoding we can simply calculate the loss function for given text label.

4.5 Recognizer

Recognizer was created by using TensorFlow library. Before we begin to assembles a computational graph we

4.5.1 Computational Graph

TensorFlow requires to assemble a computational graph which will represent the computational steps.

4.5.1.1 CTC Network

In source code we have `CTCNetwork` class representing important features of the network such as input and output dimensions, loss function or used optimizer.

The first method is `generate_placeholders`. *Placeholders* are Tensor-Flow objects able to store tensors. They don't have to be initialized and input tensors are provided during runtime. Their main purpose is for input and output values. Therefore, the method `generate_placeholders` is creating input and output placeholders for the computational graph. Input placeholder for the network is created as three dimensional array. First dimension represents batch index, second is for number of time-steps and last is for the length of acoustic vector (MFCC vector). For input is also created another placeholder of sequence length for each one on the batched sentences. Output of network is represented by a sparse placeholder because it is required by TensorFlow's CTC.

Second method `loss_funtion` creates CTC loss function inside a computational graph. We use TensorFlow method `tf.nn.ctc_loss` which takes input parameters as a label in sparse matrix format, logits which is the last layer of the network and sequence length. The TensorFlow method also performs softmax operation upon the input before applying CTC loss.

The third method is `train_optimizer` is defines the used optimizer in the graph. Optimizer is performing some type of gradient descent algorithm to minimize the error on the loss function. There are many optimizer to choose from but currently the recognizer uses one of the most popular and universal optimizer in deep learning which is *AdamOptimizer*.

Another method is `decoder` which decodes predicated sentence from outputted probabilities using argument of input sequence length placeholder and output from last layer. It uses TensorFlow method called `tf.nn.ctc_greedy_decoder`. The same output can be decoded also by using `tf.nn.ctc_beam_search_decoder` but it is little slower than the greedy decoder.

Last method is `compute_label_error_rate` which takes parameter as a decoded sparse label and computes its label error rate.

4.5.1.2 LSTM CTC

Class `LSTMCTC` extends from the `CTCNetwork` class and it defines the inner structure of the network. The constructor sets number of layers, hidden neurons, input dimension and the size of acoustic vector.

The class has method `define` which creates the part of the computation graph. It calls parent method for generating placeholders. Creates LSTM cells using `tf.contrib.rnn.LSTMCell` method for all layers then we stack the cells into multilayer RNN networks with method `tf.contrib.rnn.MultiRNNCell`, the stacked network is used in method `tf.nn.dynamic_rnn` which finalizes it

29

with input placeholders. The method `define` returns the output layer of the network.

4.5.2 Training

Training phase of the recognizer is implemented in file `train.py` by method `train_network` which takes dataset and config reader object. The method first has to read the hyperparametrs of the network from the config reader and then the computational graph is constructed using the `LSTMCTC` methods.

In TensorFlow the computation on created graphs are performed inside a `tf.Session()`, thus the training phase is happing inside the session where we loop thorough all the training epochs. In the epoch we train RNN on all training data which are provided using dataset object's method `next_batch`. To run the c we will use function `session.run(fetches, feed_data)`. The *fetches* will be graph operation which are responsible for the training and *feed_data* are network's placeholder with assigned values from `next_batch` method in dictionary structure. Example code of running the session for backpropagation algorithm:

```
feed = {
  lstm_ctc.input_placeholder : train_x,
  lstm_ctc.label_sparse_placeholder : train_y_sparse,
  lstm_ctc.input_seq_len_placeholder : train_sequence_length
}

batch_cost, _ = session.run([loss_operation, optimizer_operation], feed)
```

TensorFlow also offers a way of restoring trained networks. During a training, we may save checkpoint files with operations variables because `tf.Variable` maintains state in the graph across the computations. It's achieved by an object `tf.train.Saver()`, we either call method `save(session, checkpoint_path)` or `restore(session, checkpoint_path)`.

4.6 Robot NAO

Robot NAO is an autonomous, programmable humanoid robot and the goal is to use the implemented speech recognizer as a voice-user interface.

`ALProxy` provides remote connection to the NAO robot and gives us access to all the robot's methods. Speech recognizer will be python module running remotely and using `ALProxy` object we are able to fetch the recorded robot's sound data. The sound data will be processed by the speech recognizer and robot can react to the predicated text.

Figure 4.3: Robot NAO

Experiments

In this section we will review the speech recognizer performance. We will introduce some optimization to increase the learning model accuracy and also by tweaking hyperparameters of the network we can achieve better results.

5.1 Computing Power

Training neural networks could be considered as computational difficult problem. However, with the right hardware we can speed up the process significantly. Backpropagation algorithm is mostly about multiplying matrices and GPUs are explicitly designed to handle multiple matrix calculations at the same time, therefore it is highly recommended to use GPUs for training neural networks.

Unfortunately, TensorFlow is just limited on using NVIDIA GPUs to properly work because the python library `tensorflow-gpu` which handles the TensorFlow GPUs computations is build upon CUDA toolkit. Therefore, I will be using CPU for the experiments section as the main computational resource. Because it would not be possible to train speech recognizer on the whole *VCTK dataset*, for the experiment part I will use *Free Spoken Digit Dataset*.

The final training of the speech recognizer using *VCTK dataset* is done on *Floyd Hub* which is a commercial Platform-as-a-Service for training and deploying deep learning models in the cloud.

5.2 Dropout

Optimization of the learning model can be achieved by introducing dropout method. Simply, the dropout ignores random neurons in the layer by given probability value during a training phase. It majorly reduces overfitting on given dataset.

Hyperparameters	
number of hidden neurons	100
number of hidden layers	1
batch size	8
number of epochs	400
learning rate	0.001
dimension of acoustic vector	13
dropout	1 (N/A)

Figure 5.1: Configuration of hyperparameters

We have applied the dropout to all RNN hidden layers with the dropout probability of 0.5 which is considered as optimal in the study of *Ref*.

5.3   Cached Extracted Features

Training of speech recognition is extremely time consuming, that is the reason why we cached the extracted features.

In the initial stage of creating a computational TensorFlow graph, we perform preprocessing and feature extraction on all the given training dataset. The acoustic vectors and labels are stored as *numpy* array in the dataset class. During a training phase, whenever `next_batch` method is called, it retrieves a batch of preprocessed training data and no extra computation upon the data is required. The duration of training phase was reduced by 30% on my personal computer. However, the speed up can certainly very on different hardware configurations.

In TensorFlow, this approach is not considered the most efficient, because TensorFlow library provides `tf.data` module which can optimize the given training dataset for computational graph.

5.4   Training on Digits

Training of the implemented speech recognizer on *Free Spoken Digit Dataset* was first performed with the configuration of hyperparameters shown on Figure 5.1.

Validation of the speech recognizer performance is evaluated using label error rate which is the *Levenshtein distance*, the minimum number of single-character edits. On Figure 5.2 is shown how label error rate is decreasing though the training process. The final result error rate with the configuration 5.1 is 3%.

Unfortunately, the label error rate on Figure 5.2 is calculated on the training data. The right approach would be to calculate the performance of the
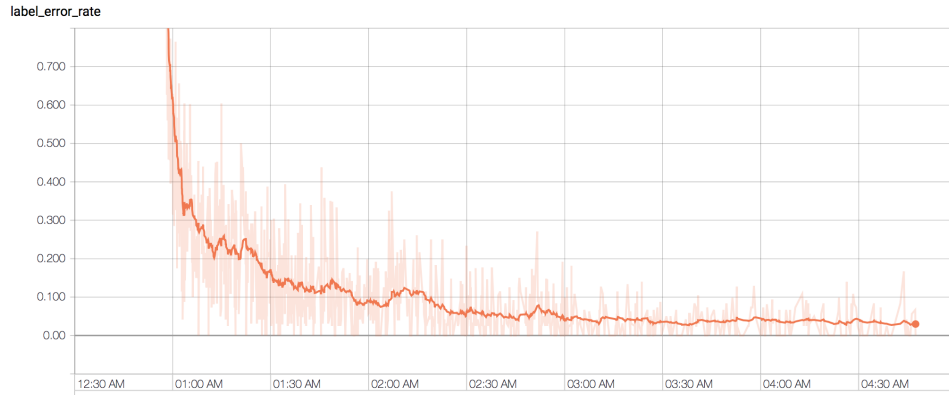
Figure 5.2: Learning Error rate on Digits for configuration on Figure 5.1

Hyperparameters	
number of hidden neurons	100
number of hidden layers	2
batch size	8
number of epochs	400
learning rate	0.001
dimension of acoustic vector	13
dropout	0.5

Figure 5.3: Configuration of hyperparameters

learning model on validation dataset. That is the reason why we get such a low error rate, the model is overfitted.

To fight against overfitting, we will apply dropout method on new network configuration shown on Figure 5.3. We have added another hidden RNN layer and dropout with probability of 50%. Introducing another layer should improve the learning model performance, but our model was overfitted and dropout is constantly devaluating the performance. As shown on Figure 5.4, the network error rate is around 5% which is worse performance than the first trained model with just one hidden layer.

This outcome could be expected because the the first model was overfitted and the second model was always devaluated by dropout. However, during the validation phase on learning model, the dropout has to be deactivated. Whenever we tried to run the second trained model without the dropout, we decreased the error rate from 5% to 1.5%. Even running the model on validation dataset, we perform around 2% error rate.
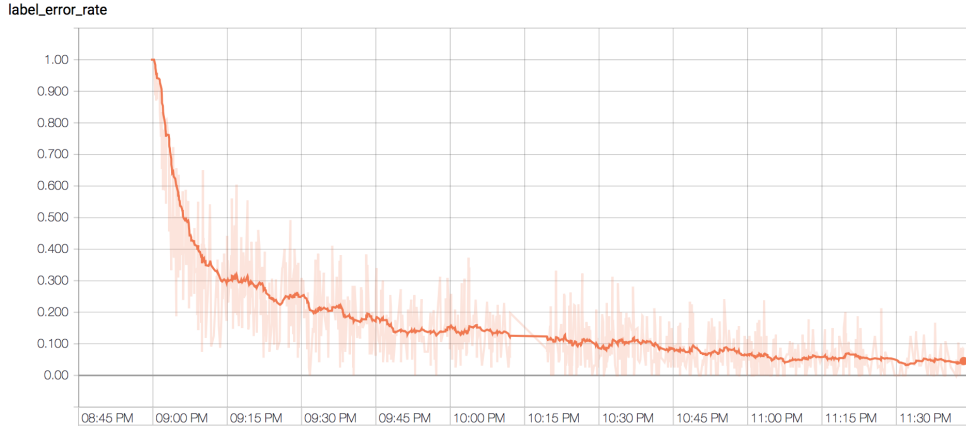
35

label_error_rate

Figure 5.4: Learning Error rate on Digits for configuration on Figure 5.3

Hyperparameters	
number of hidden neurons	100
number of hidden layers	3
batch size	64
number of epochs	1000
learning rate	0.001
dimension of acoustic vector	13
dropout	0.8

Figure 5.5: Configuration of hyperparameters

5.5 Training on VCTK Corpus

VCTK corpus was primarily created for HMM-based speech recognizers with hand picked sentences to contain all the phonemes. Unfortunately, without further exploration and deeper understanding of the VCTK corpus, we cannot simply created validation dataset using for example `train_test_split` method from *sklearn* library which we used for the *Free Spoken Digit Dataset*. Our workaround to the validation dataset problem is to instead of using different sentences, we will exchange the speakers. Even though, it is not ideal, we will verify if our speech recognizer is speaker independent.

Training speech recognizer on whole VCTK Corpus would take a couple of days and that is why we are going to limit ourself on some fraction of it. We have trained the main learning model of the end-to-end speech recognizer on just three speakers and they will be changing during the process. We have empirically chosen these configuration of hyperparameters described on Figure 5.5.

On the Figure 5.6 is illustrated how label error rate was decreasing during a training process achieving as low as 5%. However, we have not exchanged

36

the speakers so we could consider that speech recognizer is speaker dependent and dropout was set to just ignore 20% of neurons so overfitting is highly possible.
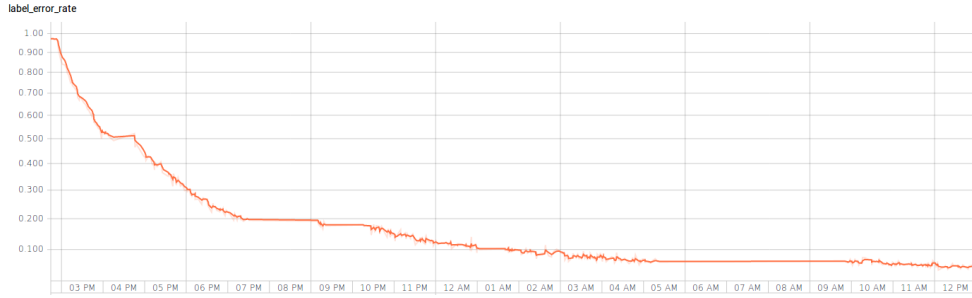

Figure 5.6: Learning Error rate on VCTK for configuration on Figure 5.5

We have increased the dropout to 50% and the label error rate spiked up to 23% as shown of Figure 5.7. Afterwards, we deactivated the dropout and label error rate immediately decreased from 11% to 5% as shown in the middle of Figure 5.7. To validate if the data are overfitted, we started to exchange speakers and the dataset error rate was fluctuating around error rate of 3%.
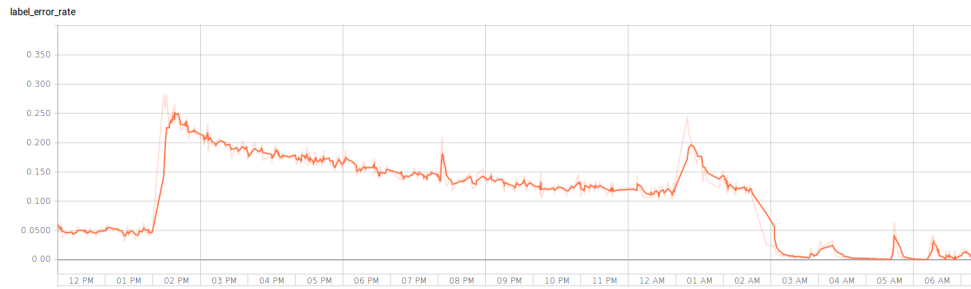

Figure 5.7: Continuation of Learning Error rate on VCTK for configuration on Figure 5.5 with dropout reduced to 0.5

Even though, it seems that recognizer is performing very well, we don't have the right validation data and that's why we cannot state the general recognizer performance. Also to use this the speech recognizer in day to day tasks, it needs to be trained on more complex dataset.

Conclusion

The goal of the thesis was to get familiar with the speech recognition field and implement speech recognizer using neural networks which would be used as voice-user interface on Robot NAO.

We have covered the topics of artificial neural networks and recurrent neural network, and we explained how backpropagation algorithm works during a training phase. Afterwards we explored speech recognition architectures and explained how speech signals is modified for the purposes of speech recognition.

The implemented solution of end-to-end speech recognizer is build upon recurrent neural networks with LSTM neurons, CTC loss function and speech signal features are extracted using MFCC. The recognizer was firstly trained on Free Spoken Digit Dataset where we achieved error rate of 3% but the model was overfitted. We have tried to tweak the hyperparameters for better performance and use dropout as optimization technique against overfitted. We have successfully lowered the error rate on 2%. The main end-to-end speech recognizer which will be used in Robot NAO, used VCTK Corpus as training dataset and achieved error rate of 3%.

In future work we want to finish the integration of implemented speech recognition with Robot NAO and train the RNN on more complex speech corpus with deeper network. We would like to improve the recognizer by using bidirectional recurrent neural networks and trained on large dataset with properly prepared validation and test data.

References

[1] *A Gentle Introduction to Exploding Gradients in Neural Networks*. URL: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/ (visited on 09/27/2017).

[2] Dario Amodei and Rishita Anubhai. "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin". In: *CoRR* abs/1512.02595 (2015). arXiv: 1512.02595. URL: http://arxiv.org/abs/1512.02595.

[3] S B. Davis and Paul Mermelstein. "Mermelstein, P.: Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. IEEE Trans. Acoust Speech Signal Processing 28(4), 357-366". In: 28 (Sept. 1980), pp. 357–366.

[4] Y. Bengio, P. Simard, and P. Frasconi. "Learning Long-term Dependencies with Gradient Descent is Difficult". In: *Trans. Neur. Netw.* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227. DOI: 10.1109/72.279181. URL: http://dx.doi.org/10.1109/72.279181.

[5] Denny Britz. *Attention and Memory in Deep Learning and NLP*. URL: http://www.wildml.com/2016/01/attention-and-memory-in-deep-learning-and-nlp/ (visited on 04/27/2016).

[6] Denny Britz. *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs*. URL: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[7] Denny Britz. *Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients*. URL: http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/.

[8] William Chan and Navdeep Jaitly. "Listen, Attend and Spell". In: *CoRR* abs/1508.01211 (2015). arXiv: 1508.01211. URL: http://arxiv.org/abs/1508.01211.

[9] Kendra Cherry and Steven Gans. *How Many Neurons Are in the Brain?* URL: `https://www.verywellmind.com/how-many-neurons-are-in-the-brain-2794889`.

[10] Chung-Cheng Chiu and Tara N. Sainath. "State-of-the-art Speech Recognition With Sequence-to-Sequence Models". In: *CoRR* abs/1712.01769 (2017). arXiv: `1712.01769`. URL: `http://arxiv.org/abs/1712.01769`.

[11] Haytham Fayek. *Speech Processing for Machine Learning: Filter banks, Mel-Frequency Cepstral Coefficients (MFCCs) and What's In-Between.* URL: `http://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html`.

[12] Filipe. *Text Generation with Recurrent Neural Networks (RNNs).* URL: `https://blog.paperspace.com/recurrent-neural-networks-part-1-2/` (visited on 01/04/2017).

[13] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks.* Vol. 385. Jan. 2012.

[14] Alex Graves et al. "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks". In: *Proceedings of the 23rd International Conference on Machine Learning.* ICML '06. ACM, 2006, pp. 369–376. ISBN: 1-59593-383-2. DOI: `10.1145/1143844.1143891`. URL: `http://doi.acm.org/10.1145/1143844.1143891`.

[15] Sepp Hochreiter and JÃŒrgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[16] Andre Holzner. *LSTM cells in PyTorch.* URL: `https://medium.com/@andre.holzner/lstm-cells-in-pytorch-fab924a78b1c` (visited on 08/10/2018).

[17] Zohar Jackson. *free-spoken-digit-dataset.* URL: `https://github.com/Jakobovski/free-spoken-digit-dataset`.

[18] Ralph Jacobson. *2.5 quintillion bytes of data created every day. How does CPG Retail manage it?* URL: `https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/`.

[19] A.J. Jerri. "The Shannon sampling theoremâIts various extensions and applications: A tutorial review". In: 65 (Dec. 1977), pp. 1565–1596.

[20] Martin Kleinsteuber. *Hidden Markov Models for Speech Recognition.* URL: `http://recognize-speech.com/acoustic-model/hidden-markov-models` (visited on 12/02/2014).

[21]   Martin Kleinsteuber. *Mel-Frequency Cepstral Coefficients*. URL: `http://recognize-speech.com/feature-extraction/mfcc` (visited on 11/25/2014).

[22]   Martin Kleinsteuber. *Preprocessing*. URL: `http://recognize-speech.com/preprocessing` (visited on 12/28/2014).

[23]   Z. C. Lipton, J. Berkowitz, and C. Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: *ArXiv e-prints* (May 2015). arXiv: `1506.00019 [cs.LG]`.

[24]   Ye Ma et al. "Reconstruct Recurrent Neural Networks via Flexible Sub-Models for Time Series Classification". In: *Applied Sciences* 8.4 (2018). ISSN: 2076-3417. DOI: `10.3390/app8040630`. URL: `http://www.mdpi.com/2076-3417/8/4/630`.

[25]   Karl N. *Google voice search: faster and more accurate*. URL: `https://ai.googleblog.com/2015/09/google-voice-search-faster-and-more.html` (visited on 09/24/2015).

[26]   Karl N. *Speech Recognition: You down with CTC?* URL: `https://gab41.lab41.org/speech-recognition-you-down-with-ctc-8d3b558943f0` (visited on 10/14/2016).

[27]   *Neuron*. URL: `https://en.wikipedia.org/wiki/Neuron` (visited on 08/10/2018).

[28]   R. Pascanu, T. Mikolov, and Y. Bengio. "On the difficulty of training Recurrent Neural Networks". In: *ArXiv e-prints* (Nov. 2012). arXiv: `1211.5063 [cs.LG]`.

[29]   Mirco Ravanelli. "Deep Learning for Distant Speech Recognition". In: *CoRR* abs/1712.06086 (2017). arXiv: `1712.06086`. URL: `http://arxiv.org/abs/1712.06086`.

[30]   Suman Saksamudre and Ratnadeep Deshmukh. "Comparative Study of Isolated Word Recognition System for Hindi Language". In: 4 (July 2015), pp. 536–540.

[31]   *Sampling Frequency and Bit Resolution for Speech Signal Processing*. URL: `http://iitg.vlab.co.in/?sub=59&brch=164&sim=474&cnt=1` (visited on 08/10/2011).

[32]   Christos Stergiou and Dimitrios Siganos. *NEURAL NETWORKS*. June 29, 2007. URL: `https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Introduction%20to%20neural%20networks`.

[33]   *Understanding FFTs and Windowing*. Tech. rep. National Instruments. URL: `http://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf`.

[34] Jake VanderPlas. *Understanding the FFT Algorithm*. URL: https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/ (visited on 08/28/2013).

[35] Paul Werbos. "Backpropagation through time: what it does and how to do it". In: 78 (Nov. 1990), pp. 1550–1560.

[36] Junichi Yamagishi. *English Multi-speaker Corpus for CSTR Voice Cloning Toolkit*. URL: http://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html (visited on 04/27/2010).

[37] Pablo Zegers. *Speech Recognition Using Neural Networks*. Dec. 2001, p. 25. URL: http://www.informatik.uni-ulm.de/ni/staff/FSchwenker/lit/papers/zegers98speech.pdf.

[38] Ying Zhang. "Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks". In: *CoRR* abs/1701.02720 (2017). arXiv: 1701.02720. URL: http://arxiv.org/abs/1701.02720.

[39] Huiting Zheng, Jiabin Yuan, and Long Chen. "Short-Term Load Forecasting Using EMD-LSTM Neural Networks with a Xgboost Algorithm for Feature Importance Evaluation". In: *Energies* 10.8 (2017). ISSN: 1996-1073. DOI: 10.3390/en10081168. URL: http://www.mdpi.com/1996-1073/10/8/1168.

[40] Jianqin Zhou. "On discrete cosine transform". In: *CoRR* abs/1109.0337 (2011). arXiv: 1109.0337. URL: http://arxiv.org/abs/1109.0337.

[41] Hafidz Zulkifli. *Understanding Learning Rates and How It Improves Performance in Deep Learning*. URL: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.

Acronyms

ANN Artifical Nerual Network

RNN Recurrent Nerual Network

CTC

MFCC

SR

GPU

CPU

Contents of enclosed CD

README.md .... the file contents description
BP_Adam_Zvada.pdf .... the thesis text in PDF format
bp_text/ .... the directory of the thesis
speech-recognition/ .... the directory of the speech recognizer
    src/ .... the directory of source codes for speech recognizer
    audio_numbers/ .... the directory of audio naumbers dataset
    trained_models/ .... the directory of pretrained models
    __main__.py/ .... project python main
    requirements.py/ .... python file with project requirements
    README.md/ .... repozitář Piper CI web