TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Margus Baumann 162732IABM

IDENTIFICATION OF FOREIGN LANGUAGE ACCENT FROM SPEECH USING NEURAL NETWORKS

Master's Thesis

Supervisor: Tanel Alumäe
Senior Researcher

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Margus Baumann 162732IABM

VÕÕRKEELE AKTSENDI TUVASTAMINE KÕNEST KASUTADES NÄRVIVÕRKE

Magistritöö

Juhendaja: Tanel Alumäe
Vanemteadur

Tallinn 2018

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Margus Baumann

07.05.2018

Abstract

This thesis investigates possibilities using neural networks for classifying speaker's native languages based on their accents. Accents are deviations in pronunciations that can cause tasks like speech recognition to produce inaccurate results. Predicting accents and identifying specific regions of speech they occur can be used for scientific research and real-life applications.

Proposed neural network models are with recurrent and convolutional neural network architectures. Experiments for accents identification are carried out with different implementations of selected architectures. Models are trained using supervised learning where data is labelled with speaker's native languages.

Different data representations are used in experiments. Addition to the classical data transformations with filterbanks and MFCC, a pretrained bottleneck features extraction model is used to transform the data.

Method for extracting accents specific regions in speech is proposed and visualized using transcripts alignments. Attention mechanism is used to highlight specific timesteps.

Experiments made with the Estonian Foreign Accent Corpus (EFAC) proved that recurrent and convolutional neural networks are able to achieve as good or even better results, compared to baseline system results.

This thesis is written in English and is 48 pages long, including 7 chapters, 21 figures and 7 tables.

Annotatsioon

Võõrkeele aktsendi tuvastamine kõnest kasutades närvivõrke

Käesolevas magistritöö eesmärgiks on kasutada tehislikke närvivõrke kõnelejate emakeele tuvastamiseks läbi aktsendi. Aktsendi all mõeldakse peamiselt häälduse eripärasid ja mitte teistlaadi kõnelemist, nagu on näiteks murded või inglise keelt rääkivate piirkondade kohaliku keele eripärad. Erinevates keeletöötlusega kokku puutuvates valdkondades saab aktsendi tuvastamise metoodikaid ja tulemusi kasutada kõne paremaks tuvastuseks ja uurimistööde sisendina. Keele uurimise seisukohast on tähtis teada, et milliste emakeelte rääkijatel on teatud häälikute hääldamine keeruline konkreetses võõrkeeles. Sellist infot saab kasutada näiteks paremaks võõrkeele õppeks.

Töös kasutatakse erinevaid närvivõrkude lahendusi, mis on võimelised heliandmeid töötlema. Närvivõrkude arhitektuuride hulgast on välja valitud rekurrentne ja konvolutsiooniline närvivõrk. Konvolutioonilist võrku on reeglina kasutatud just pilditöötluses, kuid käesolevas töös rakendatakse seda ka heli teisendustele. Võrgu õpetamiseks kasutatakse juhendatud õpet, kus andmed on varasemalt käsitsi klassifitseeritud vastavalt kõneleja emakeele järgi.

Närvivõrkude treenimiseks on vajalik viia helifailid kujule, mida on võimalik arvutisüsteemidel matemaatiliselt töödelda, samas kodeerides endas vajalikke tunnuseid kõne ja aktsendi tuvastamiseks. Sellisteks meetoditeks on näiteks filtripangad ja mel-sageduse kepstri kordajad (MFCC), mis muundavad heli arvuti jaoks samalaadsele kujule nagu inimesed seda tajuvad. Aktsendi tuvastuseks osutusid mel-sageduse kepstri kordajad liiga pealiskaudseteks. Nende edasine transformatsioon, kasutades eeltreenitud kõnetuvastuse ülekandeõppe-põhiseid pudelikaela tunnuseid (*bottleneck features*), andsid klassifitseerimisel kõige paremaid tulemusi.

Aktsendi eristamiseks ja kuvamiseks on võimalik kasutada rekurrentsele mudelile lisatavat tähelepanu mehhanismi, mis võtab sisendina erinevad ajahetked ja leiab kaalutud ajahetkede tähtsused. Tähelepanu mehhanismi sisemise tulemuse ja

kõnetuvastuses eraldatud transkriptsiooni joondamisel on võimalik leida, millised häälikud või nende kombinatsioonid on emakeelele omaste tunnuste määramisel olulised.

Magistritöös tehtavad katsetused on tehtud kasutades Eesti võõrkeele aktsendikorpust (EFAC). Tehtud eksperimendid kinnitavad, et erinevate tehislike närvivõrkude kasutamine aktsendituvastuseks on õigustatud tänu nende mitmekesisusele ja võimele klassikaliste meetoditele sarnast ja isegi paremat täpsust saavutada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 48 leheküljel, 7 peatükki, 21 joonist, 7 tabelit.

List of abbreviations and terms

AID	Accent Identification
ANN	Artificial Neural Network
API	Application programming interface
BILSTM	Bidirectional Long-Short Term Memory
CNN	Convolutional Neural network
CPU	Central Processing Unit
DCT	Discrete cosine transform
DEV	Development
DNN	Deep Neural Network
FBANK	Log Mel-scale filter banks
GPU	Graphics Processing unit
LID	Language Identification
LSTM	Long Short-Term Memory
MFCC	Mel-Frequency Cepstral Coefficients
MLP	Multilayer Perceptron
NLP	Natural Language Processing
NN	Neural Network (model)
RAM	Random-Access Memory
ReLU	Rectifying Linear Unit
RNN	Recurrent Neural Network (model)
TDNN	time-delay neural network
TPU	Tensor Processing Unit
TTÜ	Tallinn University of Technology
UBM	Universal Background Model
1D	One-dimensional
2D	Two-dimensional

Table of contents

List of figures

List of tables

1 Introduction

Speech contains more information than just linguistic content. Speech also includes clues to speaker's age, gender, social background. Additionally, it gives information about accent which is a manner of pronunciation specific to particular individual or nation.

People can have deviations in pronunciations when speaking in a non-native language. These deviations can be labelled as accents. We need to differentiate between accent that focuses on pronunciations and a dialect that in addition can encompass grammar, vocabulary and other regional or social attributes. In our research we will be focusing primarily on accents.

This thesis will investigate using neural networks for automatic foreign language accent identification. We fill focus on the following research questions:

- What kind of acoustic features are most suitable for accent classification;
- Which neural network architectures result in most accurate accent classification models;
- What kind of regularization mechanisms to use to deal with data sparseness and overfitting issues;
- How to use the attention mechanism of the recurrent neural network to analyse and interpret its predictions.

For humans ignoring speech irregularities caused by accents is done automatically and mostly with a high degree of accuracy. Computer systems that implement speech recognition struggle with different accents because it has no knowledge of deviations that are present in a speech which can lead to incorrect results. A dedicated system is needed to help identify those irregularities.

While conducting the research we will implement an accent identification (AID) system that is validated and performs at least as well as other implementations, such as I-vector or Deep Neural Networks (DNN).

Over the last few years different Artificial Neural Network (ANN) or just Neural Network (NN) architectures have shown good progress in helping to process Natural Language Processing (NLP) tasks like speech recognition. Recurrent Neural Network (RNN), as one of the forms of neural network is by its essence suitable for processing sequence data like sounds, video and text. RNN-s can also have different architecture like Long Short-Term Memory (LSTM).

Proposed AID research will be conducted with sequenced processing capable Neural Networks (NN) architectures:

- Recurrent Neural Networks (RNN) with modified LSTM implementation in the form of Bidirectional Long Short-Term Memory (BILSTM).
- Convolutional Neural Network (CNN) with 1D and 2D filter configurations

This thesis will cover constructing models that are trained, tuned and evaluated against Estonian language accent corpus [1]. Samples for the system will be from recorded natural speech that have consistent content texts but have different native language backgrounds. Results from different models will be compared to baseline system accuracy.

In our research we will be analysing how people with different native foreign language will pronounce same sentences and words in Estonian and try to compare them to each other and also to the baseline language. Our aim is to identify specific regions of speech deviations and compare them over different languages to find patters for classification and clustering.

The outcome of the thesis could be used in the field of scientific research and in real life applications. The trained accent classification models could be used for determining which words or parts of words are difficult to pronounce to people with certain language backgrounds. Other applications could use the proposed system for increasing the accuracy of a speech recognition system: the accent identification system could serve as a step in a speech recognition pipeline where the result of the accent identification system

could be used for selecting a speech recognition system adapted to the particular accent. Assigning automatically identified accents to speakers could be also be used for enriching the metadata of automatically transcribed audio archives. Research could also be used for improving natural speech synthesis [2].

2 Background - Theory

In this chapter we will try to give some intuitions on different neural network architectures, implementations and methods that can be used to achieve specific tasks. Also, what are the capabilities and restrictions of those implementations.

2.1 Neural networks

Artificial neural networks are a special type of computational techniques in the field of computer science and machine learning for approximating complexed functions. They have been developed from gaining inspirations from biological neural networks with addition to different mathematical learning algorithms. Artificial neural networks have been also called *"deep neural networks"* and recently have been referenced as *"deep learning"*. Although the naming's can be confusing the main idea behind the different names comes from the development of the artificial neural design and complexity of the different implementations. Deep learning is about learning data representations with more than one layer of hierarchical structure that can be implemented by artificial neural networks. Neural networks can be with many different architectures and layer designs, but in current thesis will cover the main two sequence suitable architectures: convolutional neural networks and recurrent neural networks. Multilayer perceptron's as the simplest neural networks will be also explained as it is used in other implementations. The idea behind all of them is that they consist of computational units that are linked with weights between the hidden layers.

The main feature behind neural networks is that it can find representations and has iterative learning capabilities. This thesis will be implemented by using supervised learning where the network is trained with labelled data. Mostly used learning algorithm today, in supervised learning, is backpropagation algorithm with gradient descent optimization algorithm [3, p. 15]. Multiple gradient decent optimization algorithms have been developed to help speed up the learning process: Adam, RMSprop, Adadelta etc [4, p. 35].

Implementing neural networks have been difficult over the years and there have been some obstacles that needed to be overcome:

- Finding efficient learning algorithms led to back propagation with gradient decent
- Lots of data, especially labelled data has been relieved by large digital labelled datasets
- Need for computational power has been relieved by further development of RAM, CPU, GPU and the use of distributed and parallel computing

The need for fast methods and accurate results for various data processing tasks, that can be implemented on many devices have increased the research effort and development of different machine learning methods in which neural network-based architectures have become a significant alternative. Many advances have been made with different structures of neural networks and the increased of computational power with using GPU, tensor processing unit (TPU) has helped to achieve this goal.

Neural networks in a supervised learning context where learning is done with labelled data, is good at two tasks:

- Regression
- Classifications

Regression is for tasks where predicted value is scalar numeric value and classification, when the output is a probability distribution over set of predefined classes.

Neural networks are used in many industries for different applications and all sorts of problem solving like computer vision, speech, natural language processing, robotics and predictive business processes.

Task that need to be solved can have data in various formats, like sound, images or raw sensor outputs. When selecting the right neural network structure, we need to know what kind of architectures are suitable for different data.

Current thesis about sound processing thus we have selected network types that have the ability to process sequenced data that we will cover in the next few chapters. Mainly focusing on recurrent and convolutional implementations.

2.2 Multilayer perceptron

Multilayer perceptron (MLP) is a simple feedforward network that has at least two layers and an input layer. Layers that are aligned between input and output are also called hidden layers [5, p. 527], see Figure 1. Hidden layers are not observable and their values are determined by the model by which features representations are most relevant [3, p. 6]. The idea behind feedforward network, is that data from examples are passed forward in a network and the results is compared to the actual values. Loss function is used to calculate the difference and backpropagation is used to update the weights.

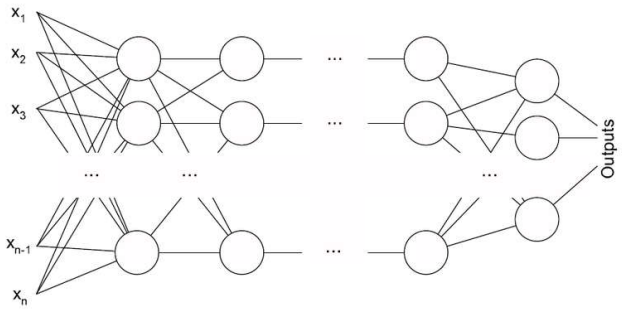


Figure 1 Feedforward network with multiple hidden layers.

MLP network consists of cells or neurons that are aligned in a specific manner, usually by layers and each layer neurons are connected to previous layer active neurons. Each layer consists of one or more neurons that take incoming initial values and multiply by corresponding weights that are then summed together and a bias is added. Calculated value is then feed into an activation function, usually a nonlinear function is used, that produces a neuron output value for next layer neurons. This process is done for every active neuron in a layer and for every layer from input to output.

Each layer (h) in a network can be also interpreted as a n-dimensional vector where dimensions are the units in a layer. Connections (W) between layers can be written as a matrix combining input values (x) to layer cells. A fully connected layer is a

transformation from one vector to another by implementing a vector-matrix multiplication by a weight matrix and by applying elementwise nonlinear function (g) [4, pp. 42-47]. In order to represent complex functions a nonlinear activation function is needed to transform from simple linear model (equation 1), also called as perceptron, to a multi-layer perceptron (equation 2):

$$h = x\,W + b \qquad (1)$$
$$x \in \mathbb{R}^{d_{in}}, \quad W = \mathbb{R}^{d_{in} \times d_{out}}, \quad b = \mathbb{R}^{d_{out}}$$

$$h^1 = g^1\,(x\,W^1 + b^1) \qquad (2)$$
$$h^l = g^l\,(h^{l-1}\,W^l + b^l)$$
$$x \in \mathbb{R}^{d_{in}}, \quad W = \mathbb{R}^{d_{in} \times d_{out}}, \quad b = \mathbb{R}^{d_{out}}, \quad l - layer$$

Layers between output and input are referred as *hidden layers* and they always have some sort of nonlinearity function that is applied. There are several common nonlinearities that is also called as activation functions: sigmoid, tanh, rectified linear unit, etc. The list is not complete and many variations exists like leaky rectified linear unit. There is also no specific domain, where each of the nonlinearities should be used, but recently there have been good results when using rectifying linear unit (ReLU) [4, p. 45]. Sigmoid and tanh is also widely used specially in the end of the model to squeeze the values to appropriate range.

Sigmoid (σ) is also known as the logistic function that outputs a value in the range between (0,1) and is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad (3)$$

Hyperbolic tangent (tanh) results on the value range between (-1, 1):

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \qquad (4)$$

Rectified linear unit ReLu clips values if x < 0:

$$ReLU(x) = \max(0, x) \tag{5}$$

Last layer in a network is the output layer. Depending of the model task, regression or classification, different kind of activations can be used. In the case of regression, we can use linear function. If we need to limit the range of the output then a non-linear, like sigmoid, can also be used. Classifications on another hand needs to transform the output into probability estimations. For binary variables the output is a probability between 0 and 1, which can be produced by using sigmoid function. Multi-class predictions need some sort of function to represent the probabilities of a distribution [3, p. 81]. Softmax function is a technique that forces predictions (ŷ) to be positive values and sum to 1 [4, p. 24]. Softmax:

$$softmax(x)_{[i]} = \frac{e^{x_{[i]}}}{\sum_j e^{x_{[j]}}} \tag{6}$$

$$\hat{y}_{[i]} = \frac{e^{(x\,W+b)_{[i]}}}{\sum_j e^{x\,W+b_{[j]}}}$$

Training neural network requires us to define a loss function, also known as a cost or error function (L). The training objective is to minimize the loss when predicting in respect to dataset true value. Loss function returns a scalar value when comparing network output (ŷ) to expected output (y) [4, p. 46].

$$L(\hat{y}, y) \tag{7}$$

Regression models can use L1 (Mean absolute error) or L2 (mean squared error). For classification tasks categorical or binary cross entropy can be used.

Multilayer perceptron-based models are able to predict and classify in a wide range of different domains and with different representations of data. In some representations, like sequenced data, simple neural network architecture lacks the complexity to predict in the context of sequence order dependencies. Recurrent and convolutional networks have the

same principles as multilayer perceptron, but with more complexed architectures and they can be better at specific feature extractions.

2.3 Recurrent Neural Network

Multilayer perceptron network structures have limitations. Sequenced represented data is one of the form that simple neural networks have difficulty finding representations. For processing sequenced data like timeseries, a more suitable network model is required. Recurrent neural networks (RNN) have been developed for this specific purpose. The idea behind RNN is that it can process sequenced data by taking each timestep and previous timestep value with memory capabilities and extract sequenced representations.

Recurrent neural networks are deep in time and space by having hidden states that span thru previous hidden states and multiple layers, if stacked on top as regular feedforward layers in an MLP architecture [6].

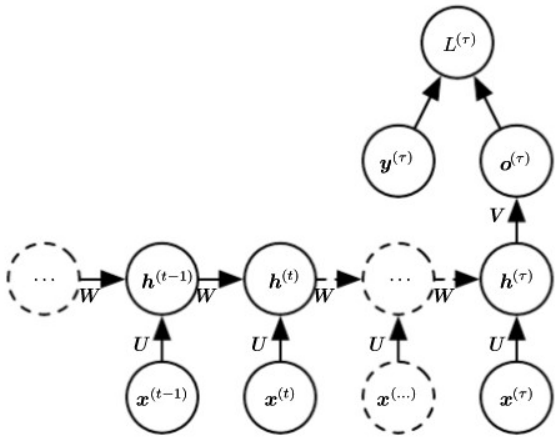


Figure 2 Unrolled recurrent neural network [3, p. 382].

A standard RNN, see Figure 2, computes for every timestep (t) a hidden (h_t) and an output (y_t) vector given an input (x_t) [6]:

$$\begin{gathered} h_t = H(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ y_t = W_{hy}h_t + b_y \\ H - sigmoid, t - time \end{gathered} \tag{8}$$

This kind of sequence computations and keeping all representations of all the timesteps by sharing the same hidden matrix W. This can lead to vanishing and exploding gradient problems [3, p. 290]. In simple feedforward network backpropagation in done from output to input, but for recurrent network the process is somewhat extended with backpropagation through time. Because with recurrent network is sharing the same weights matrix thru every step then learning can cause gradient to grow or decay exponentially. Exploding gradients can be prevented by gradient clipping, but this does not help with vanishing ones [3, p. 415]. Alleviating long term dependencies can be achieved with gated architectures of RNN: long short-term memory (LSTM) and gated recurrent unit GRU. In current thesis, only LSTM are explained and used.

LSTM, see Figure 3, is a special RNN implementation that tries to manage long-term dependencies by implementing gates: input, output and forget. These gates decide what information will be used and what will be forgotten.

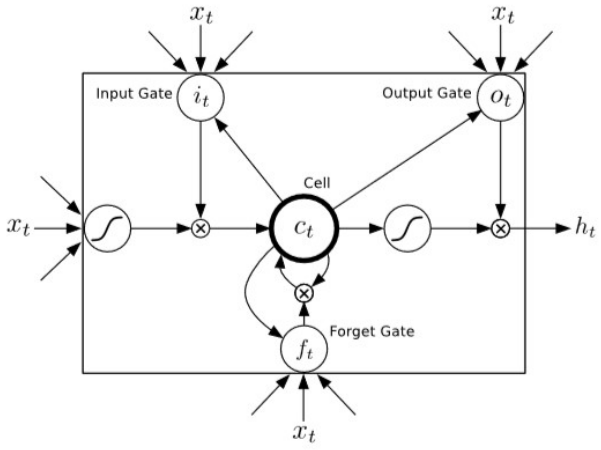


Figure 3 LSTM cell structure [6].

LSTM cell can be implemented by following functions [6]:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \quad (9)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \quad (10)$$

$$c_t = f_t c_{t-1} + i_t \, tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad (11)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o) \quad (12)$$

$$h_t = o_t \tanh(c_t) \quad (13)$$

$i - input\ gate, f - forget\ gate, o - output\ gate, h - hidden, c - cell\ state$

Working with sequenced data like speech and text, not only previous context may be in interest, but also future ones. By implementing timesteps from 1 to T and T to 1, in separate layers and feeding them to the same output, the model is able to access context information in both directions. This kind on implementation is called a bidirectional RNN, see Figure 4Figure 1, and it also can be implemented on LSTM [6].

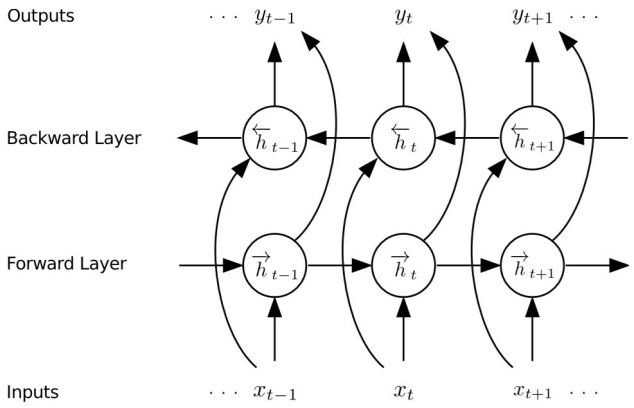


Figure 4 Bidiectional recurrent layers [6].

Using LSTM or other RNN cells can be computationally complex and therefore a more efficient architecture is needed that can process sequence data and achieve similar good result as recurrent networks. Convolutional networks are one alternative.

2.4 Convolutional Neural Network

Convolutional networks have primarily been used for computer vision and image processing. CNN-s are good at fixed sized grid like structures. Although CNN is preliminarily used with image data, which is a grid of width * height, it is also used with fixed length sequence data, like fixed length padded timeseries, with dimensions of features * timesteps.

CNN architecture has different layers: convolutional, max pooling (down sampling), up sampling, deconvolutional, average pooling. Up sampling and deconvolutional is mostly used in advanced architectures and in unsupervised learning tasks not covered in this

thesis [2] [7]. Convolutional layers can be stacked as layers in a network with many combinations. Typical implementation of stacked CNN layers includes: convolutional, nonlinear activation and pooling, see Figure 5. This ordering of layers is repeated multiple times which is followed by fully connected feed forward layers.

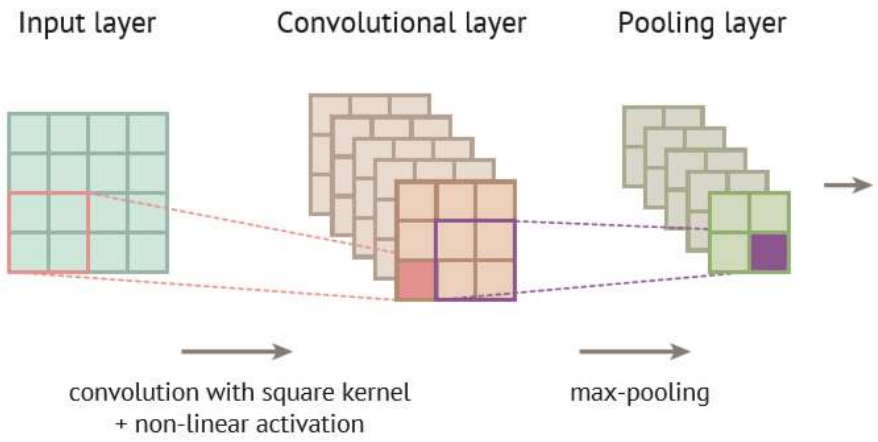


Figure 5 Traditional implementation of convolutional architecture [8].

In many cases CNN is not used as a standalone network. Its layers are only useful as sub-structure extractors that can then be feed forward to other types of architectures that can be used for predicting and classifying more complex relationships [4, p. 152]. The last few layers in CNN models are usually regular fully connected multilayer perceptron's. CNN can also have recurrent networks attached or included to help better predictions for sequenced features [9].

There are two main convolutional operations: two-dimensional (2D) and one-dimensional (1D). When dealing with images, then 2D is usually used. With sequenced data like sound and text 1D convolutions are usually used [4, p. 152]. The main difference between 1D and 2D is the size and movement of the convolutional filters. The 1D implementation has filter size spanning over all features in a selected timesteps and therefore only moving over sequenced representations, see Figure 6.

Current thesis will cover using both kind of convolutional implementation, 2D and 1D, for speech and sound identification [8].

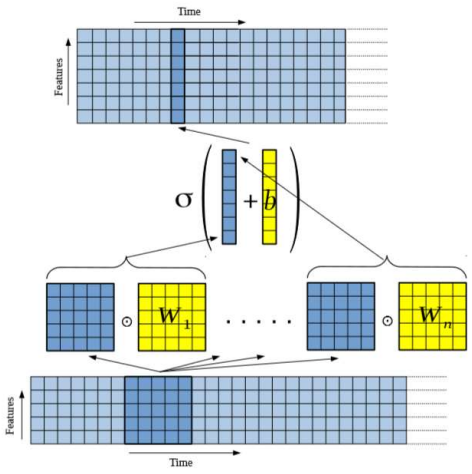


Figure 6 1D convolution over time [10].

Convolution architectures are able to extract specific features and detect objects, regardless of the position in the grid. Mainly detected features are edges and corners. Convolutional layers filters or hidden units are not connected to all of the previous layer input but limited to only a small part that is called a *receptive field* [8]. The weights of a filter are shared when filter is applied to the previous layers input and the result is a *feature map* or a filter [8]. Convolutional layer can have multiple filters that simultaneously extract features. As shows in Figure 7, stacking multiple layers on top of each other enables higher layers to use extracted features to construct complexed patterns that can then be used to be used in classification or other tasks the model requires.

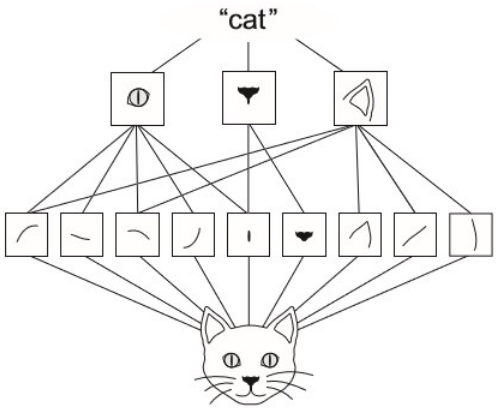


Figure 7 Extracting hierarchical features with multiple layers [11].

Appling filters to input space causes the field size to be reduced. Padding can be used in order to keep the original shape and size. *Padding* and *stride* are two main parameters used when manipulating the size of the output. Usually zero padding is used so the filter kernel size wouldn't affect the output size [3, p. 357]. Stride parameter is used to increase the length of the receptive field sliding step over input area. Commonly only stride 1 is used, although using stride larger the one can lead to more spatial feature extractions and is equivalent to using max-pooling [3, p. 350] [10]. In some cases when stride is used then max-pooling is even not necessary [12].

There is also a method called *dilatation* that has recently been used to reduce the complexity of the convolutional model. Google's Wavenet is one example of using dilated convolutions to increase the receptive fields, with just a few layers stacked together [2].

CNN can have one or more pooling layers, additionally to convolutional layers. It is used when reduction of the dimensionality is needed. Most common pooling layer operations are average and max pooling. They are used to extract maximum or average output values from a neighbourhood and therefore making representations become invariant to changes [3, p. 342]. When the input changes by a small amount then pooling, especially max-pooling output stays the same. Pooling operations causes information to be lost by only taking the specific output values and this may lead to information loss.

Pooling layers can be also used as an alternative to flatten operations to turn convolutional net spatial feature maps into a vector that can then be used to connect to feed forward network [11, p. 321].

Convolutional architecture provides an alternative or a supplement to recurrent model and because the different methodology of feature extractions or representation learning. Combining those models in different combinations have shown good result [13].

2.5 Regularization

Supervised machine learning is about approximating the target function to map the inputs to the outputs and in this process learning general attributes. Generalization is one of the most important aspect of training a model. In the training phase, only limited number of examples are provided and from them, the model must be able to generalize important

features in order to correctly predict on data that has not been seen before. Generalization can be seen from validation data that is not used in training.

In the process of training, relevant features and patters are discovered until generalization flattens, as seen from validation metrics: accuracy and loss. If the loss from development data starts to decrease and accuracy doesn't improve, then model has shifted from underfitting to overfitting and has started to learn some training data specific features that are not relevant to new data [11, p. 104]. Model has then moved from finding general features to specific mappings.

Overfitting can be prevented by regulating the network and the learning process or by increasing the training data. It all depends on the speed the learning begins to overfit. When overfitting starts too fast then it may indicate to oversized model and the simplest solution would be to increase the size. If the model starts to overfit in the later stages then it may need some constraints.

2.5.1 Dropout

In order to prevent overfitting, a method for making random part of the model inactive, has been successful for regulating the network to find new pathways and generating new representations. Some parts of the networks are randomly with some probability shot down in order the network to find new pathways, see Figure 8.

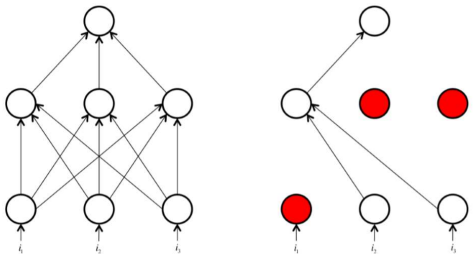


Figure 8 applying dropout [14].

Dropout is only used during training and not in testing when it is set to 0. Selected probability value is determined by the network, data and observations.

2.5.2 Weight Decay L1 & L2 Regularization

Preventing fast overfitting can be done by a method called weight regularization and its implemented by putting constraints to the model. Constants can be added by decreasing its weights values with a cost. In current thesis only L2 regularization or weight decay is added. It penalizes model for high parameter weights if it is high enough, and L2 loses its effect when the value is close to zero [4].

2.5.3 Confidence Penalizing

The model's objective or loss function can also be regularized. Models that are built for categorical classification can have categorical cross entropy loss with confidence penalty.

Confident predictions are related to low entropy output distribution. Network can be over confident when some classes have too large probabilities and therefor starts to overfit. Applying confidence penalty as a regularization can prevent biased distributions and can help to lead to better generalization by smoother output distributions [15]. Adding confidence penalty to output distribution can be done by applying negative entropy to the negative log-likelihood [15]:

$$L(\theta) = -\sum \log p_0(y|x) - \beta H(p_0(y|x)) \tag{14}$$

Penalty strength is controlled by parameter β. Assigning beta value to 1 has shown speech recognition tasks with TIMIT corpus the best results [15].

2.5.4 Class Weights

Training data is not always equally distributed and this can cause the model start to be biased to specific training data. Counteracting biased classifiers can be done by assigning class weights to counterbalance and teaching the network to equalize learning by classes in order to pay attention to underrepresented classes. Class weights needs to be inversely proportional to class frequency. In order to prevent training from preferring dominant classes they must be scaled using a parameter [16]:

$$\widehat{N} = \frac{N}{|L|} \tag{15}$$

$$\alpha_i = \frac{\widehat{N}}{N_i}$$

$$N - examples, \qquad |L| - unique\ class, \qquad N_i - class\ examples$$

Taking the average number of examples per classes and dividing by specific class results a scaler for under and over represented classes.

2.6 Weights Initializations

Initializing network internal parameters (weights and biases) is important because of under- and overfitting the network during learning process. Each layer in a network can be initialized with specific weights and biases. Kernel initializer is used to initialize weights and Bias initializer to set bias values. Weights are usually initialized using randomization in a specific range in between 0 and 1 or -1 and 1. Random weights are important because of in the learning phase weight are being updated and if they are too small or too big then this may cause learning to be hard by vanishing or exploding gradients [3, p. 304].

2.7 Bottleneck Features

Bottleneck features (BNF) refers to a network architecture where one layer, usually in the latter part of the model, has a reduced dimensionality. It has been shown to compress info from mapping input to output, making the system more robustness to noise and overfitting [17]. This kind of model can have two outputs. Full model is used when training the model from end-to-end. Second output is on bottleneck layer where output from this layer is used to extract new features with specific dimensionality. The idea is that bottleneck outputs have a good representation of the learned hidden features with reduced dimensionalities.

Bottleneck features can be trained by speech recognition, mapping audio representations to phonemes. This is usually done with very large speech corpus. Running new data thru the model with BNF layer as output, we are able to extract specific feature mapping or speech dynamics that can help on classification tasks like accent identification.

Bottleneck features can be used to outperform alternative i-vector based systems [18].

2.8 Attention with Context

Attention mechanisms are an addition to recurrent network. Finding a representation in a RNN is usually done in the last step of the cell output. Not Depending on the type of RNN cell used, the extracted features are usually transformed into a fixed length representation that can then be used with classification or regression. Gaining insight into what hidden states were important in the contexed of the whole input, can help the model improve accuracy. Attention mechanisms have been helpful with this kind of tasks [19] [20].

Attention can help models extract better hidden features and also help us get better insight into which part of the input was important to impact the result. It is implemented by introducing as attention mechanism which takes input at each timestep h_{it} and applies a single layer MLP to extract its hidden value u_{it} [19]:

$$u_{it} = tanh(W_w h_{it} + b_w) \tag{16}$$

Hidden value importance is then measured with similarity of hidden context vector u_w to calculate normalized importance weight α_{it} (attention context weight) [19]:

$$\alpha_{it} = \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)} \tag{17}$$

Output vector value s_{it} of attention, can be calculated by [19]:

$$s_{it} = \sum_t \alpha_{it} h_{it} \tag{18}$$

Attention mechanism, see Figure 9, weight α_{it} is what makes creating context output at each timestep possible by distributing importance over features.

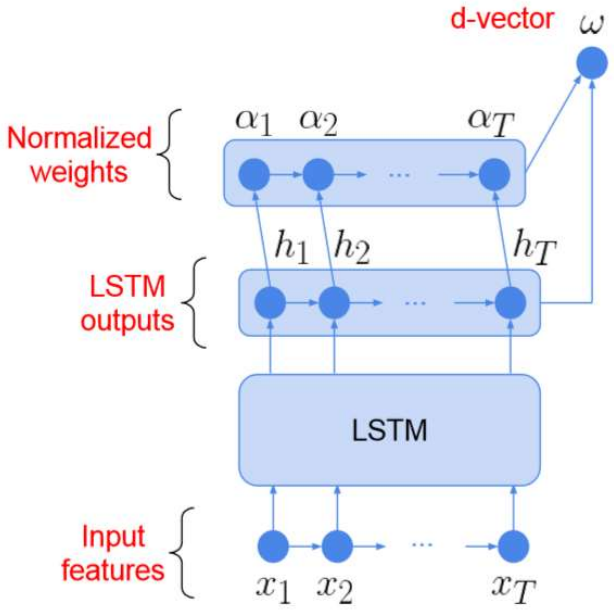


Figure 9 LSTM with attention [21].

Attention with context should be used with bidirectional implementations of RNN, giving insights into both directions and therefore widening the context window.

Multiple Attention implementation have been used for different tasks: attention-based encoder-decoder for speech recognition, computer vision with optical character recognition [20]. Attention mechanism is a simple but effective method to extract important features and also gain insights in what part of the input is important. It can also be used width visualization of the internals of the model. Code for attention implementation can be found in Appendix 5 – Python Code for Keras Additional Layer - Attention With Context.

2.9 Data Distribution

Building a model and using some data to train and test will require some pre-processing. Usually data is split into 3 different sub datasets [11, p. 97]: training, validation (development) and test. Training and validation set is preliminary used in training and test is held-out and only used in the final testing on how much the model has learned to

generalize. Depending on the size of the whole dataset, splitting can be made with different distributions. Smaller datasets are usually split with rule 80/10/10 (train, validation, test). If the dataset is very large, maybe containing hundreds of thousands or even millions of samples, then taking huge number to validate and test is not feasible and can cause training to suffer.

Data should always be distributed with balanced representations and shuffled before splitting, especially when dealing with classifications. If data has 3 classes (A, B, C), then following the rule 80/10/10, data distribution should be:

- Training (A - 80%, B – 80%, C-80%)
- Development (A - 10%, B – 10%, C-10%)
- Test (A - 10%, B – 10%, C-10%)

Sometimes this kind of even class distribution of data is not always possible. When dealing with sound and human speech where you need to move the entire speaker data, so that the speaker wouldn't occur in different datasets. Depending on the task at hand, validation and test datasets can sometimes contain different samples of representations in classes or even have some classes missing. This kind of mixed distributions is called imbalanced classes. In some cases, imbalanced data can lead even better overall results, because of the datasets are differently composed and when tuning the model, you can't predict the test set.

There is also a notion of information leaking when tuning model's configuration parameters, called *hyperparameters* [11, p. 97]. If you select hyperparameters based on development set, then you are also actually leaking information about test set, because development and test have similar distributions in classes and therefore model will learn what classes are more important and what are not. This will lead to wrong conclusions that the model generalizes well on new data, when actually it is not.

Training set, as the biggest distribution, should always contain as much data as can be given, because neural networks always needs lots of data to prevent model from overfitting. More data prevents the model from learning the examples and is always a good alternative to regularization. It is important that train dataset also contains all of the

different classes representation to help achieve better training results. Model can only generalize on data it has seen, not on what it has not.

The Validation set is also called the *development or dev set*, because it is used in the development of the model. Validation set is extracted from the training or test set do be used when tuning the model. Training data is not sufficient to do hyperparameters search. Mainly because you can't predict overfitting. Validation set is used to measure the model efficiency on new data and with this knowledge tune its hyperparameters.

Test dataset is held-out from building and training phase. Only when a finished model is selected then test data it is introduced and if parameters were correctly selected then the accuracy of the test should be similar to dev set.

2.10 Sound Data Transformations

Sound data can be presented in many formats. Neural networks require the data to be transformed into numeric values that can then be used to do calculations. In native formats, sound files are needed to be transformed into some sort of state that can be presented as numeric values. This kind of formats need to contain some sort of representations of linguistic content that is needed to do mapping from input to output.

Different formats may retain different hidden representations, like information on devices used in recording, speaker's peculiarities, background environment, emotions etc.

Methods like MFCC and FBANK [22] have been developed that can be applied to reduce the complexity and filter out features not relevant or may even result to false reasoning.

Mel-Frequency Cepstral Coefficients (MFCCs) is one of the representation a sound can be transformed into. It has been widely used for sound and music modelling and speech recognition [23]. Converting waveform to MFCC [23]:

1) Convert to Frames – divide into 20ms statistically stationary sections with Hamming window (smooths out edges)

2) Take Discreate Fourier Transform (DFT) on each frame to retrain only the logarithm of the amplitude spectrum

3) Mel-scaling and smoothing - group spectral components into smaller number of bins (ca 40) where lower frequencies are more important.

4) Discrete cosine transform (DCT) – Karhunen-Loeve (KL) transform approximation to obtain lesser (ca 13) features for each frame.

Log Mel-scale filter banks energies (FBANK) is another implementation for sound transformations that is used as an alternative to MFCC [22]. The transformations are similar to previous method but with applying fixed number the filterbanks and without the DCT. Filterbanks have been also called to be motivated based on human hearing and how we perceive specific sound signals.

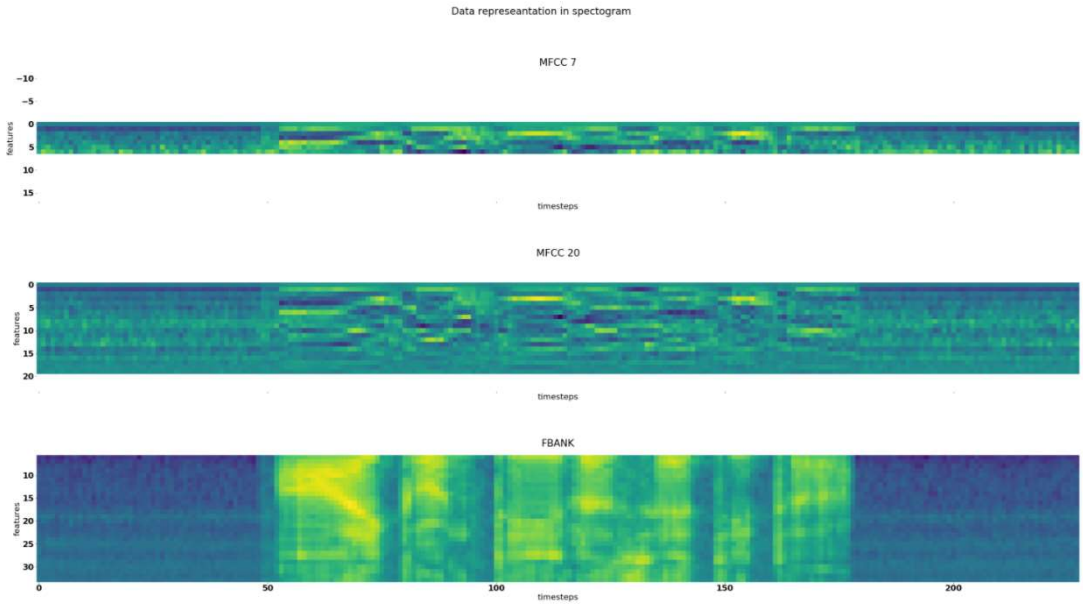


Figure 10 Data representation comparison.

When plotting data from FBANK and MFCC as shown in Figure 10, words in utterance are more visually distinguishable in filterbanks energies spectrogram then they are in two implementations of MFCC. Also, the transformations between frequencies are smoother.

2.11 Keras Framework

Keras is a high-level API library, written in python [24]. It enables users to use different backends like Tensorflow, Theano or CNTK to develop flexible and modular deep neural network models. Keras can be used to build models from simple architecture (*Sequential model*) to more complex ones (Model class with *functional API*). Keras creator Francois Chollet has released book "Deep learning with Python" [11] that has good description on how basic supervised machine learning methods can be implemented by using Keras as high-level API.

Specific steps for training, validating and testing on Keras models:

1. Compile: configures the model for training with specific parameters.
2. Fit: Model training with training data set for a number of predefined epochs.
3. Evaluate: Training and tuning the model with train and development (dev) dataset. Returns metrics for accuracy and loss.
4. Predict: Testing the model with datasets. Returns predicted values from model and input.

Keras version 2.0 was used in this thesis for implementations of different models.

3 Related work

Current thesis experiments are based on previously published research on different neural network architectural implementations. This chapter give some insight into the origin of baseline model and how it is implemented. Also, the implementations that helped us to construct alternative models and data representation that will be later described in upcoming chapters.

3.1 Previous Solutions

Speech processing has evolved over time from phonotactic based approaches to i-vector implementation and currently to neural networks.

3.1.1 I-Vector

I-vectors have been used for speaker, language and accent recognition by providing a low-dimensional representation of feature vectors for recognition and classification [18].

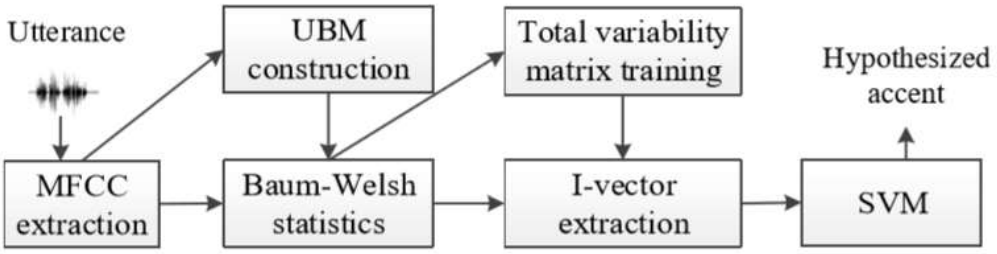


Figure 11 I-vector extraction process for accent [18].

As shown in Figure 11, i-vectors with SVM or logistics regression process can be used for accent classifications. The system first extracts data representation with MFCC extraction method, followed by Universal Background Model (UBM) construction. UBM is then used for Baum-Welch statistics. Total variability matrix is trained and i-vector is extracted as coordinates in the reduced total variability space [18].

I-vector is extracted to the entire utterance and is a fixed length representation [25]. This can lead to latency in LID tasks.

3.1.2 Phonotactic Based Approaches

Phonotactic system is a conventional phoneme recognition with Language Model (PRLM) system that used to build phoneme grids from phoneme posteriors and derive expected n-gram counts from them [17]. Those counts are then modelled with language model techniques, i-vector extraction and SVM. For speech recognition it's important to use context dependent phoneme as targets [17].

Phonotactic approaches can also be used for accent recognition by the frequency of use of certain phonetic sequences [18].

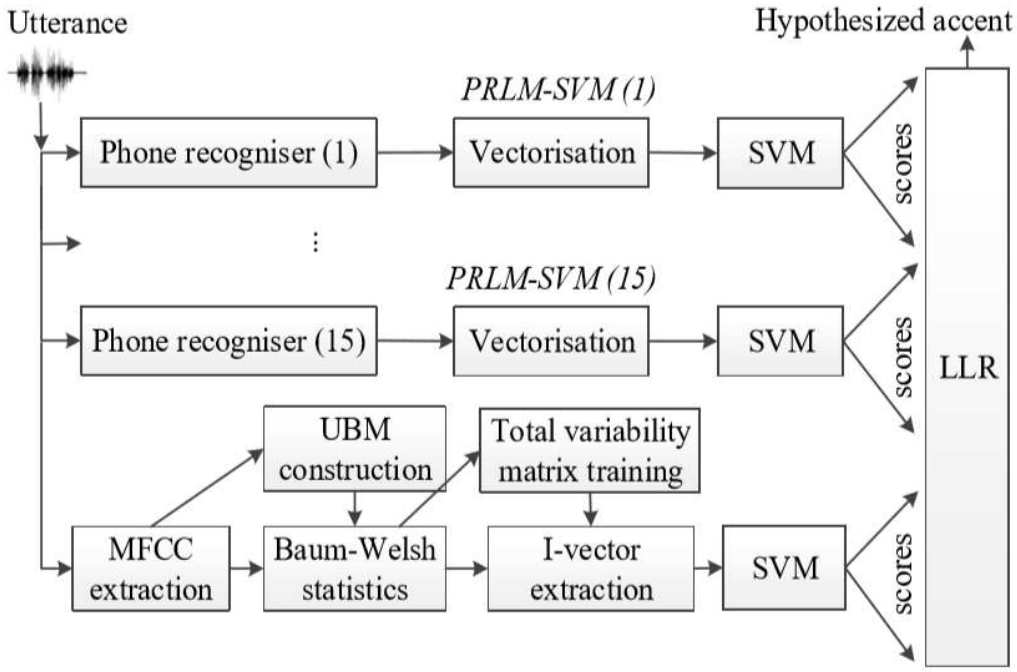


Figure 12 Combining Phonotactic with i-vectors for accent classification [18].

Fusing scores, shown in Figure 12, and described in the research of "Identification of British English regional accents using fusion of i-vector and multi-accent phonotactic systems" [18], i-vector and multi-accent phonotactic have shown better result and faster computations compared to other previous systems.

3.1.3 LSTM and CNN Systems

Recurrent neural network (RNN) architectures have been used for NLP and LID task and have noted to outperformed conventional approaches even with short utterances [25]. Main two implementations are GRU and LSTM. Sequenced based architectures have abilities to process time sequenced data and for every timestep extract features. Those values can then be weighted and most important features extracted by a process called attention mechanism as described in "Attention-Based models for text-dependant speaker verification" [21].

RNN are not the only architecture that is able to process sequential data. Convolutional models have been used for many different speech recognition tasks. Usually 1D convolutions with different size receptive fields is used because its ability to extract features from timesteps as described in "Convolutional Neural Networks with Data Augmentation for Classifying Speakers' Native Language" [10].

Fusing recurrent and convolutional architectures have shown better results than DNN, LSTM and CNN [13]. Combining two architectures together can be done by parallel computing [13] or stacking them on top [9] as shown in Figure 13.

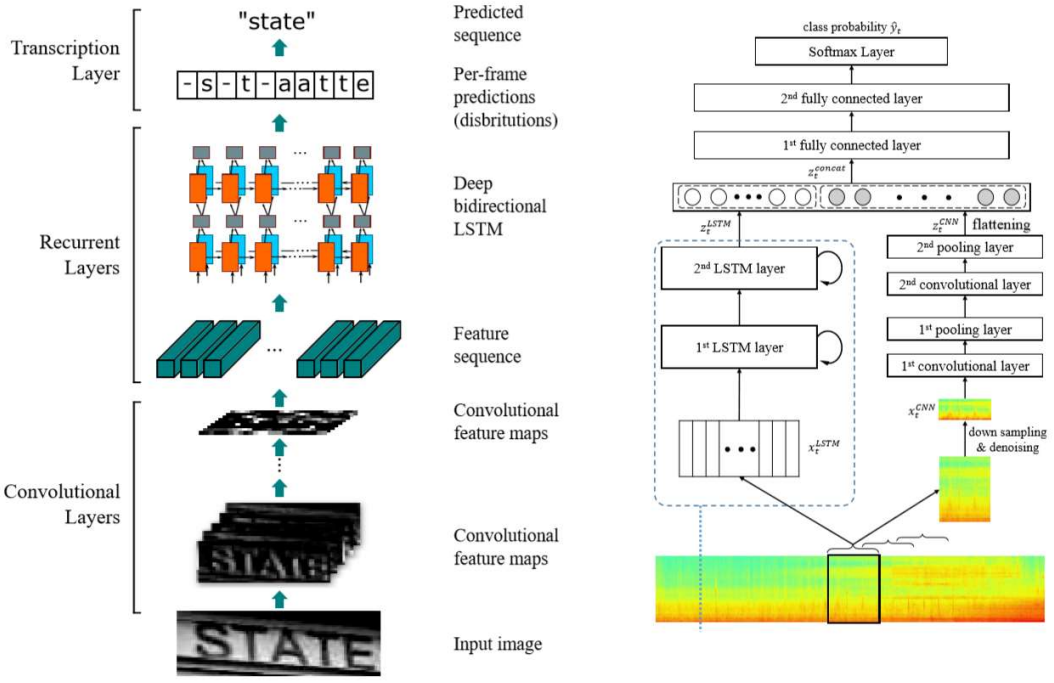


Figure 13 Stacking and parallel methods for fused networks [13] [9].

3.2 Bottleneck Feature Extractor

Extracting special features from data can be done by pretrained network that has already learned special features. Model that has a bottleneck feature already built in can be used to train to extract those features. The bottleneck feature extractor used in current thesis is trained on around 220 hours of Estonian speech recognition training data. The data originates from various domains (broadcast news, broadcast conversations, lectures and conference speeches, studio-recorded spontaneous dialogues), mostly collected at Tallinn University of Technology in the last 10 years.

The bottleneck feature extractor is a time-delay neural network (TDNN) [26], trained as a speech recognition acoustic model using the Kaldi speech recognition toolkit [27]. The input to the neural network are 40-dimensional MFCCs, extracted every 10 ms with an overlapping frame length of 25 ms. The neural network outputs correspond to clustered phoneme states (around 4000 in total) and the model is trained to perform frame-level classification. Every frame is classified using the frame itself and window of frames around it (14 from the left and 9 from the right). Five one-dimensional convolutional layers (also known as time-delay layers) are used, each with a kernel size of 650. After the convolutional layer, a bottleneck layer is inserted to the model. The bottleneck layer is a fully connected layer, with a dimensionality of 40. The bottleneck layer is followed by another fully connected layer with a dimensionality of 650, and finally a softmax classification layer. All hidden layers use the rectified linear units as the nonlinearity.

For extracting the bottleneck features, all layers of the model that follow the 40-dimensional bottleneck layer are discarded, and the model is used as a feature extractor, converting a window of MFCC features to a 40-dimensional representation.

3.3 Baseline System

Final evaluation on current research experiments will be against a baseline system that was build, trained and tested with the same accent data as current research.

The baseline AID is based on i-vectors and logistic regression. The i-vector extractor is trained by using the training data of the accent identification corpus. Input bottleneck

features are mean-normalized using a sliding window of 300 frames. Standard shifted-delta processing [28] is used to further transform the input features. 600-dimensional i-vectors are used.

For training the baseline accent identification system, i-vectors for all training utterances are extracted. Then, a standard multiclass logistic regression classifier (also known as a maximum entropy model) is trained, using the utterance i-vectors as features and the corresponding speaker accents as labels. The accuracy of the baseline system is 73.1 % on dev and 66.8 % on test data.

4 Methodology for Accent Identification Network

In this chapter we will describe our selected models that we used in more detail, how the experiments will be conducted and describe data selection for training and validation.

4.1 Accent Audio Data

Sound data in a raw format is too complex and contains too much distracting information. Data needs to be transformed into simpler representation that is relevant to accent classification. Labelled data needs to be split to train, development and test sets, in order the experiments to be correctly performed. Additional transformations are also required for the datasets to be suitable for neural networks.

4.1.1 Audio Representation

Predicting accents in speech assumes presence of accent specific acoustic features that neural network should be able to extract [22]. MFCC, as described earlier has the most transformations to remove not relevant features. This can lead to removing or smoothing out features specific to accent identification.

Tests conducted with MFCC during this thesis led to understanding that data transformed in this manner does not generalize on new data. Test dataset result were approximately 10-15% below dev results, not depending on models used.

MFCC transformations were used by bottleneck feature extractions that transformed data to be aligned with speech recognition trained phonemes (around 4000 phonemes in total were used). Figure 14 has same utterance plotted in different format to illustrate the transformations. We believe when using bottleneck features to transform the data, timeframes are mapped from input to specific phonemes and this data is then encoded to the output.

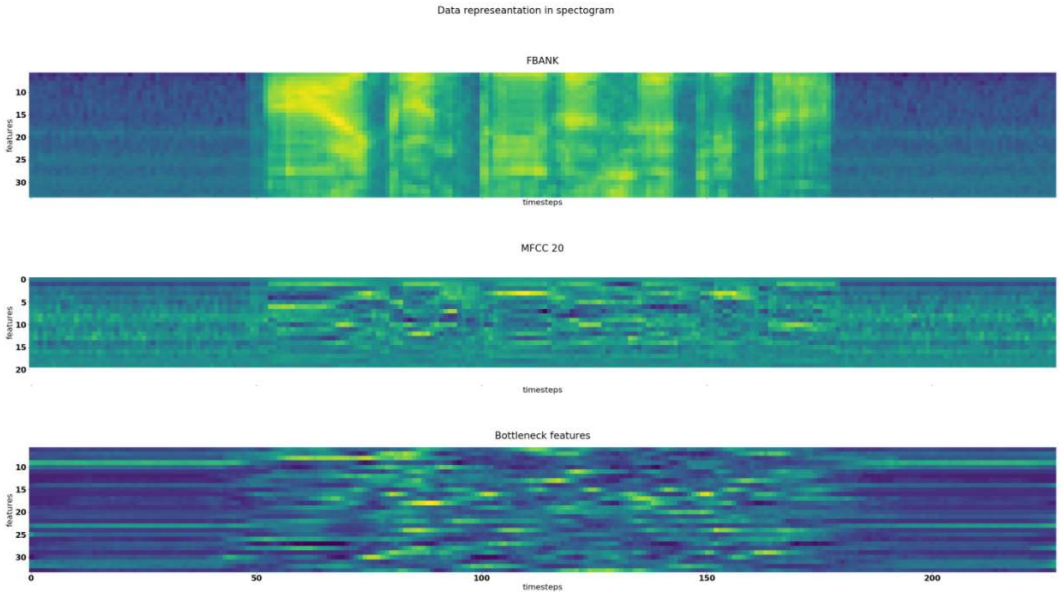


Figure 14 Spectrogram of FBANK, MFCC and Bottleneck features.

For conducting accent specific experiments, we have decided to use filterbanks and bottleneck features because we believe they will have the necessary representation for accent specific feature extractions. Data has been pre-padded with zeros to be in fixed length.

4.1.2 Estonian Foreign Accent Corpus

Estonian Foreign Accent Corpus (EFAC) was initiated by the Institute of Cybernetics, Tallinn University of Technology in 2006, with the goal of collecting non-native and native speech recordings for experimental phonetic research and related technologies development [1].

Data for accent identification research is collected from 185 speakers from 17 different native languages speakers other than Estonian, see Figure 15 and Table 1. Estonians as a reference group is represented by 21 unique speakers. The total duration of recording is approximately 80 hours [1] with 28’629 unique examples in total.

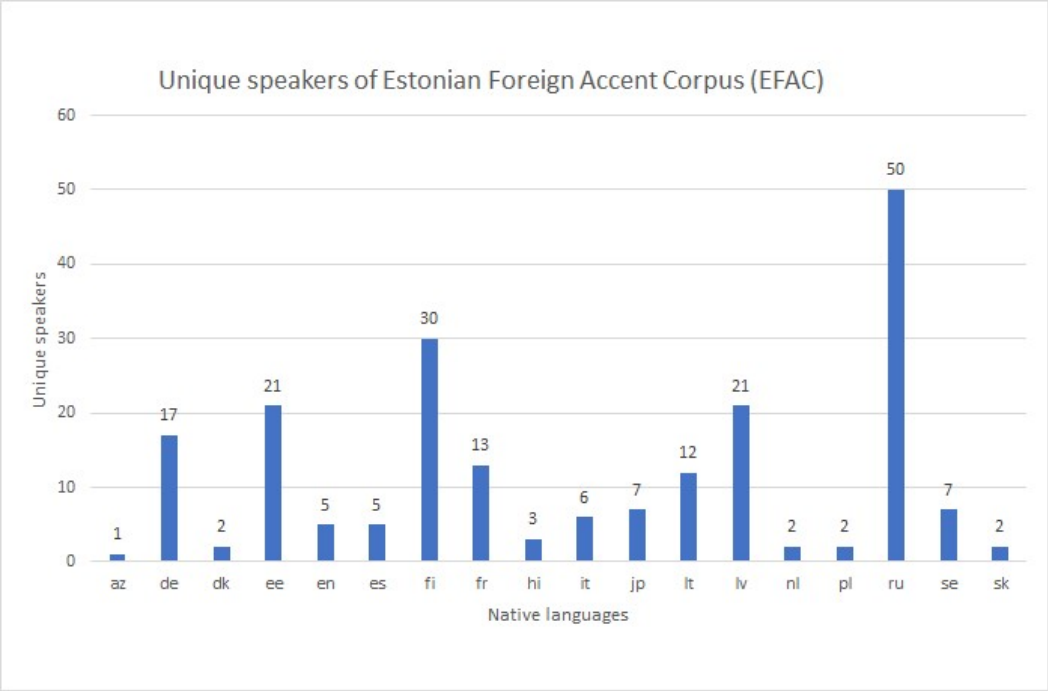


Figure 15 Native language speaker distribution.

Table 1 Unique speakers count by native languages.

az	de	dk	ee	en	es	fi	fr	hi	it	jp	lt	lv	nl	pl	ru	se	sk
1	17	2	21	5	5	30	13	3	6	7	12	21	2	2	50	7	2

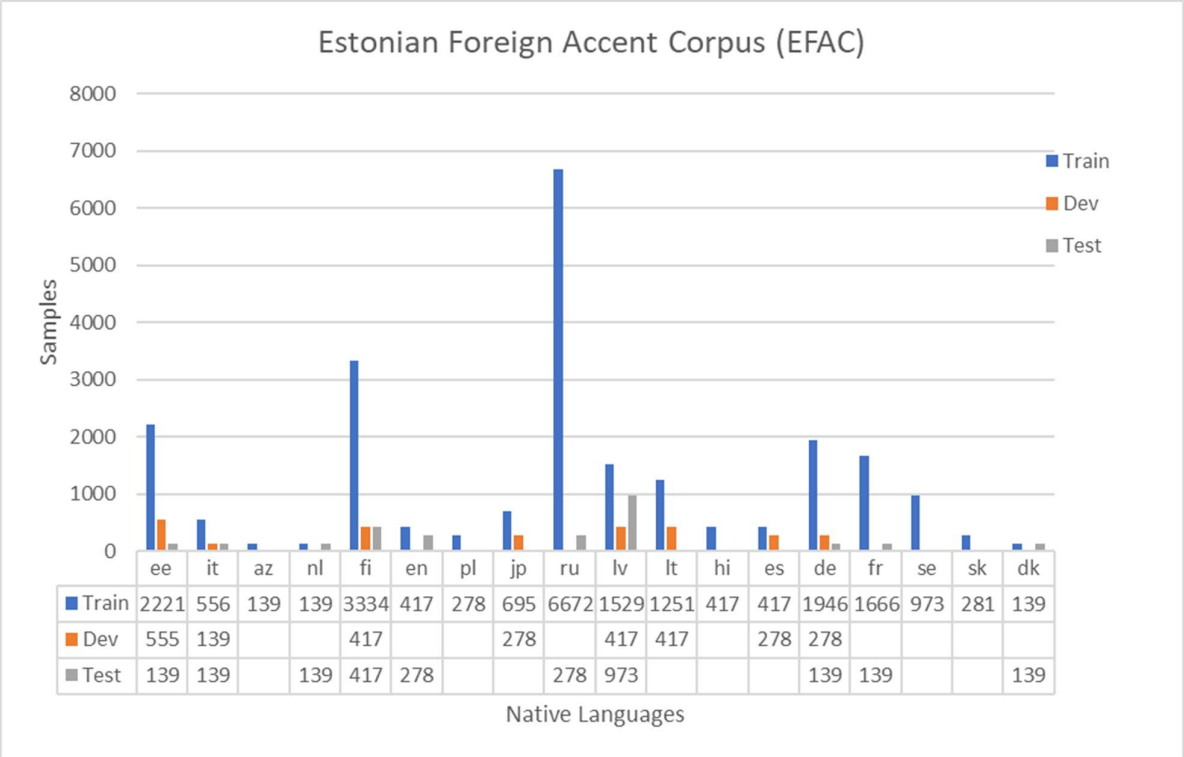
The corpus contains 139 unique sentences that users have recorded in closed studios in Estonia or in partnering universities. Users were displayed one sentence at a time and the correctness of the recordings were monitored [1]. The content of the sentences is rather simple, so that the non-native speaker can pronounce the words. The selected sentenced also cover all Estonian vowels and consonants [1].

Previous research on Estonian accent has shown that some vowels like "*ö*" and "*õ*" are acoustically very differentiable in specific languages, like Russian and therefore pronouncing words containing them could be the best candidates for accent identification [29]. Sentences like "jõulu laupäeva õhtul sõime verivorsti ja sülti" or "võtan ühe kannu külma õlut" should give us differentiable results in non-native and native pronunciations.

Data in the corpus does not include predominantly Estonian native samples, so the neural architecture will need to compare different languages and find a way to distinguish different pronunciations.

Data from accent dataset was split into multiple subsets by randomly extracting specific speakers, so same speaker would not appear in other subsets. Data was split into three sets: Training (Train) 23’070, Development (Dev) 2’779 and Testing (Test) 2’780 examples.

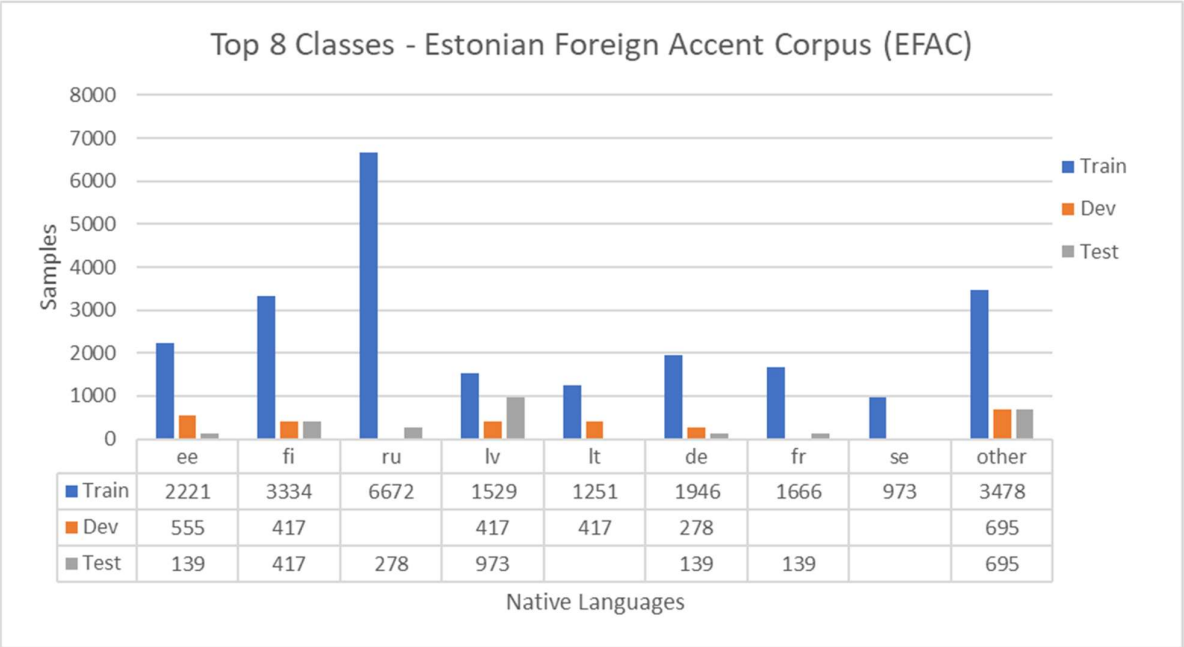
Accent corpus has sound samples that have been labelled with 18 speaker languages, as seen in Figure 16.



	ee	it	az	nl	fi	en	pl	jp	ru	lv	lt	hi	es	de	fr	se	sk	dk
Train	2221	556	139	139	3334	417	278	695	6672	1529	1251	417	417	1946	1666	973	281	139
Dev	555	139			417			278		417	417		278	278				
Test	139	139		139	417	278			278	973				139	139			139

Figure 16 Estonian Foreign Accent unique samples

Comparing and classifying so many different labels may be too difficult, considering the amount of data we have. Neural architectures need lots of data to be able to learn meaningful mappings from input to output. In order to compensate the lack of data, only top 8 of the classes are kept with their original label and the others have been re-labelled as *other*, see Figure 17 and Table 2.



	ee	fi	ru	lv	lt	de	fr	se	other
Train	2221	3334	6672	1529	1251	1946	1666	973	3478
Dev	555	417		417	417	278			695
Test	139	417	278	973		139	139		695

Figure 17 Top 8+1 classes of EFAC

Table 2 Data distribution of top 8+1 classes

Speaker native language	Train	Dev	Test
ee	2221	555	139
fi	3334	417	417
ru	6672	0	278
lv	1529	417	973
lt	1251	417	0
de	1946	278	139
fr	1666	0	139
se	973	0	0
other	3478	695	695
Samples ∑	23070	2779	2780
Splitting %	80%	10%	10%

Main reason why the *other* data has not been removed, is that this data is used as a regularization method to help the models find new pathways and achieve generalization. Another reason for keeping 8+1 classes is to prevent from false positive predictions. As seen from Figure 16, predicting classes *ru, lv* and *fi* would give us 60% accuracy on test dataset, while other classes could be classified inaccurately. This could also mean, that

the model has found a way to learn to distinguish specific languages or some other characteristics not even relevant to accent identification.

Audio data has variable length data depending on the spoken sentence text and the speaker's characteristics, like speaking tempo. Neural networks implementations by Kera require input to be fixed length if using batches in training to update the model weights and biases. Inputs need to be processed into some fixed state by cutting and padding samples to achieve same length. Current training dataset was cut to fit 80% of the individual samples, giving us max length of 560. Samples longer than this was cut and shorter ones were pre-padded with value 0 (zero). Entire dataset was cut to into fixed size of 560 x features.

4.2 Model Selection

Experiments will be made with selected architectures that have been proven to be adequate for sequential processing of sound data. As there are 2 main type of architectures: recurrent based LSTM and convolutional ones, both of them are used in current thesis. Because both architectures are using different approaches for feature extraction, a fused architecture with parallel processing is also implemented.

4.2.1 Convolutional Network Architecture

Convolutional architectures can have two different implementations for sound processing based on receptive field shape. Regular 2D with square filter sizes and rectangular 1D that spans over entire sequence input features. The main difference between the two is that 1D filters are only slided over timesteps by taking into account all that timesteps features.

Models are constructed to use stride windows for more spatial feature extractions and is replacing max-pooling layers between convolutional layers, as seen in Figure 18. Global max pooling is used to shrink the dimensions followed by dropout and a fully connected layer for predictions with SoftMax as a classifier.

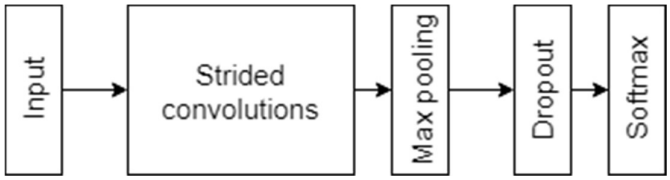


Figure 18 Proposed convolutional architecture

Learnable parametrs differes greatly between proposed convolutional networks. There are 1’250’057 in 1D and 5’741’321 in 2D. Difference comes from 2D filters ability to see only partial features in a timestep opposed to entire features.

Both convolutional implementations result will be implemented.

CONV1D model:

1. CONV1D – 128 hidden units – filter size 3 – stride 1
2. CONV1D – 128 hidden units – filter size 5 – stride 1
3. CONV1D – 256 hidden units – filter size 5 – stride 3
4. CONV1D – 256 hidden units – filter size 5 – stride 1
5. CONV1D – 512 hidden units – filter size 5 – stride 3

CONV2D model:

1. CONV2D – 128 hidden units – filter size 3 – stride 1
2. CONV2D – 256 hidden units – filter size 5 – stride 3
3. CONV2D – 256 hidden units – filter size 5 – stride 1
4. CONV2D – 512 hidden units – filter size 5 – stride 3

Specific model layers can be found in Appendix 1 – CNN with 1D and 2D in Keras.

4.2.2 LSTM with Attention

LSTM are built for sequential data processing. In combination with attention mechanism LSTM cells can be very useful for extracting and showing different important areas of the input that could then help the model to find relevant information for its goals. LSTM layers in Keras framework can also be implemented with bidirectional layer wrapper. Bidirectional enables sequences to be processed from left to right and vice versa. LSTM layers can therefore see previous and future steps, see Figure 4.

Proposed Bidirectional LSTM (BILSTM) with attention, see Figure 19, will have one LSTM layer with 32 hidden units wrapped with bidirectional wrapper, making the output size 64 extracted features. Model is also visualized in Appendix 2 – BILSTM and Attention with Context.

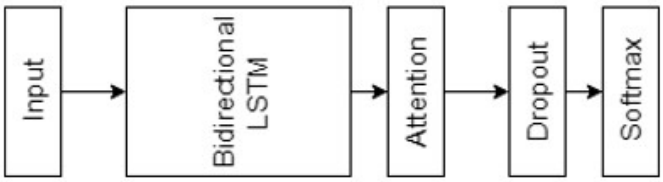


Figure 19 Proposed LSTM with attention architecture

On top the LSTM is the attention mechanism. Attention is applied by attention with context implementations described in Chapter 2.8.

4.2.3 Fused Model with CNN and LSTM

Fused models are important achievements, enabled by different libraries and the use of *Cuda* processors in GPU. Stacking or using parallel processing of models helps us build better architectures that are able to extract different types of features and by combining those result together achieve more accurate results [13].

Our proposed model, as seen in Figure 20, is based on [13] where parallel models ware used and the results were concatenated. Concatenations is used to complement the findings of those two different methods and achieve a greater feature extraction mechanism for classification. Fused model has a 2D convolutional model, described in Chapter 4.2.1, and two stacked layers of LSTM with each 32 hidden nodes, Proposed model can be seen in Appendix 3 – CNN and LSTM fused model.

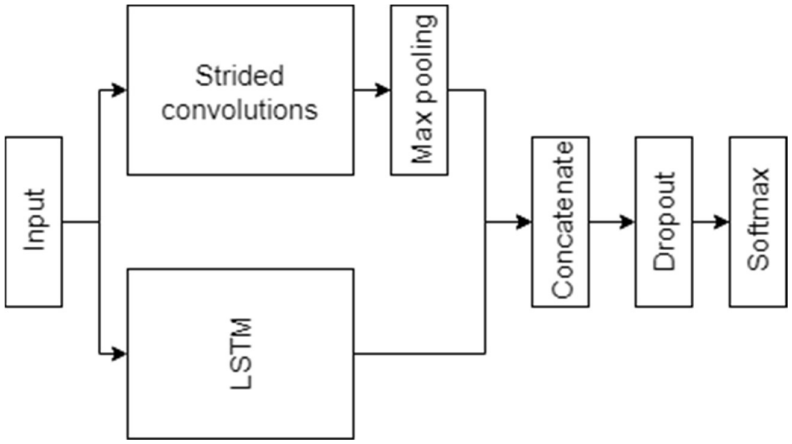


Figure 20 Proposed fused LSTM with CNN 2D architecture

Main drawback of fused model is a different speed in which models can compute their output. This means that learning is slow to do recurrent calculation slowness.

4.3 Training

All models had same L2 weights regularizes added to different layers with value set to 1e-3. Dropout layers had dropout probabilities set to 10%. Training was done with batch size 32. All training weights were saved to be used later on for predicting test set. Every model is trained with fixed 20 epochs, full iterations of all training samples. Example code for training fused model can be seen in Appendix 4 – Python Code for Training and Evaluation.

Results for each model was averaged over 5 random initializations without any predefined seed values. Best result was selected based on training cycles (*Epoch*) best development result. Models were not trained on dev dataset.

4.4 Testing

How good a neural networks works is done by measuring its generalization scope. Test set is always left out from selecting, tuning and training a model and therefore validating

against the test set gives us a good estimate on how well the model has been on generalized feature extractions.

Test results were taken based on best development result. If some model training iteration achieved better results on testing, but not on dev validation, then those were discarded.

5 Experimental Results

Timeseries data often needs models that are able to process sequenced data. In chapter 2.3 and 2.4 specific sequence capable architectures were introduced and described. Experiments are made with models:

- Recurrent model-based BILSTM with attention
- Convolutional model (1-dimensional)
- Convolutional model (2-dimensional)
- Parallel processed convolutional and recurrent based-model

Data selection based on results with different models:

- Mel-scale filter banks energies
- Bottleneck features extracted from MFCC

5.1 Implementation

Experiments were conducted using Keras library, described in chapter 2.11. Keras was initiated with *Tensorflow* as backend. Code for using Keras was written in Python. Methods or implementations not present in Keras, like *Attention* or *class weights* were added from other sources. Additional python libraries were used: sklearn, numpy, matplotlib.

Experiments did not include transformation from raw audio to used representations:

- Mel-scale filter banks
- Bottleneck features

Selecting model structure and hyperparameters is done exclusively on the vectorized scalar values for preventing hand picking of features to extract. Parameters are selected on how well the model is generalizing and not how to select or discard specific features that can be heard from the audio.

MFCC transformations was not used in this thesis results do their lack of generalizations, explained previously in chapter 4.1.1.

Experiments were conducted on GPU enabled cloud infrastructure with NVIDIA Quadro P4000 (Pascal with 1792 CUDA cores and 8GB of memory) and 30 GB of RAM memory.

5.2 Validation

Our baseline model accuracy with i-vectors were 73.1 % on development and 66.8 % on test data. Models were trained (23'070 examples) and then validated with development (2'779 examples) and evaluated on test (2'780 examples) data. Accuracies are taken by running every model and dataset combination 5 times and taking the averages of those runs. As can be seen from Table 3, regular neural networks can achieve as better results as selected baseline system.

Table 3 Experimental results for 5 iterations with 20 Epoch. Standard deviation.

		FBANK		Bottleneck	
Model name	Learnable parameters	Dev accuracy	Test accuracy	Dev accuracy	Test accuracy
CONV1D	1'250'057	68% ± 1.3	65% ± 2.6	69% ± 1.3	68% ± 3.83
CONV2D	5'741'321	59% ± 1.8	56% ± 3.2	71% ± 2.5	75% ± 3.0
BILSTM_ATTENT	23'497	57% ± 2.6	52% ± 5.2	63% ± 2.0	56% ± 3.7
CONV2D_LSTM_CONCAT	5'759'273	56% ± 2.9	58% ± 1.7	69% ± 2.1	76% ± 2.0

Bottleneck features reach better accuracy score then regular filterbanks representation. Only 1D convolutions are able to somehow extract as good result from non-bottleneck data. When comparing baseline to current results, test vs dev accuracy, it can be seen that some models that have convolutional layers are able to generalize better. They can achieve better results from test and still sustain high accuracy on dev dataset.

Training times differ significantly from selected models and by the tools used, see Table 4. Using LSTM layers in a model increases the complexity and training time compared to *Conv2D* with almost same number of learnable parameters.

Table 4 Training times for 20 Epoch

Model name	Learnable parameters	Average Time for 1 Epoch	Total Training Time 1 Epoch
CONV1D	1'250'057	00:00:18	00:06:03
CONV2D	5'741'321	00:05:06	01:42:07
BILSTM_ATTENT	23'497	00:20:44	06:54:40
CONV2D_LSTM_CONCAT	5'759'273	00:26:05	08:41:35

Best model, based on the result, is CONV2D and this model accuracy we will try to increase by additional regularization.

5.3 Loss Regularizations

To achieve even better results from the best model, chosen from previous chapter, some additional regularization techniques can be implemented. Methods described in chapter 2.5.3 and 2.5.4 are not implemented on the previous chapter models.

Regularization methods to add:

- Confidence penalizing
- Class weights

Using bottleneck features and CONV2D architecture, the model was trained again to achieve better generalizations. Additional regularization methods helped to increase accuracy of the test set, as seen from Table 5.

Table 5 Experimental results for additional regularizations with 5 iterations (20 Epoch each) compared to previous results. Standard deviation.

		Bottleneck		**Bottleneck with regularization**	
Model name	**Learnable parameters**	**Dev accuracy**	**Test accuracy**	**Dev accuracy**	**Test accuracy**
CONV2D	5'741'321	71% ± 2.5	75% ± 3.0	69% ± 1.2	82% ± 1.5

We could achieve better results on the test set but not on the development set. Dev set accuracy even decreased by few percentage. Training time accuracy and loss comparison, as seen from Figure 21, suggest that better dev and test results can be achieved with some regularization tuning and running the training process longer than 20 epochs. Class weights and confidence penalizing as seen from Figure 21, smooths out the training loss.

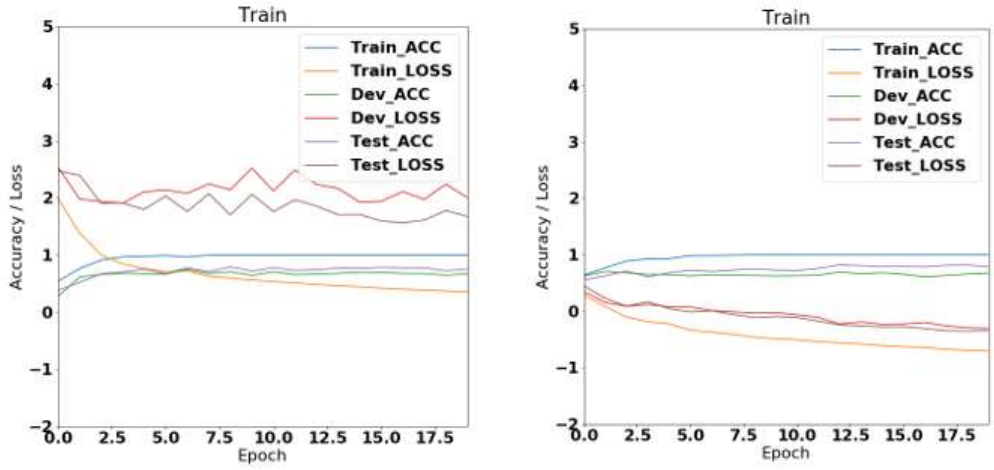


Figure 21 CONV2D validation without (left) and with regularization (right)

From Table 6, we can see that with less than 20 epochs, the model could reach almost 82% accuracy with bottleneck features. Based on the results of the maximum dev model weights, performance specific performance metrics have been calculated. *Precision* value is the number of correctly predicted labels to the total number of predicted labels to specific class. *Recall* is the number of correctly predicted labels to the total of specific class. *F1* score shows us weighted average of Precision and Recall.

Table 6 CONV2D with regularization results

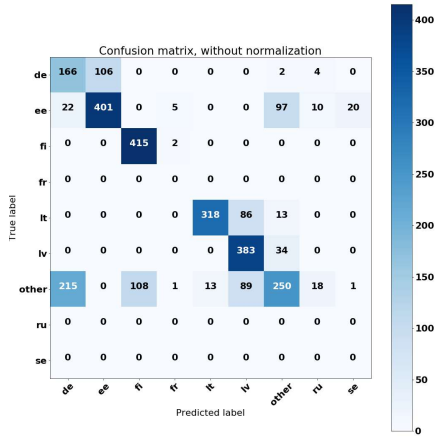
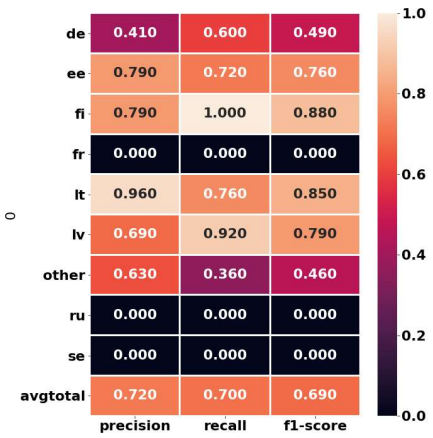
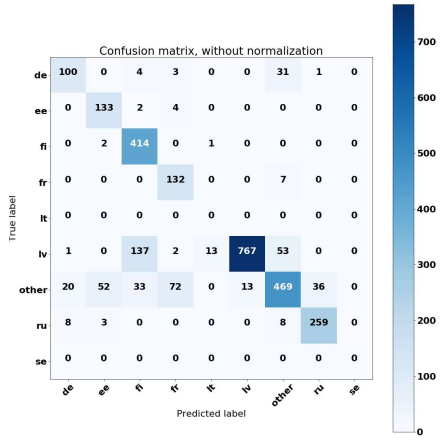
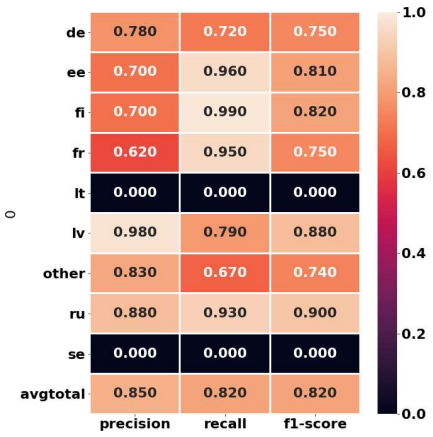
Dataset	Confusion matrix	Classification report
Dev		
Test		

Table 6 also shows us that the model struggles with speakers labelled as *other* in dev and test datasets. This is understandable because this class contains different language background speakers, explained in the chapter 4.1.2 with more detail.

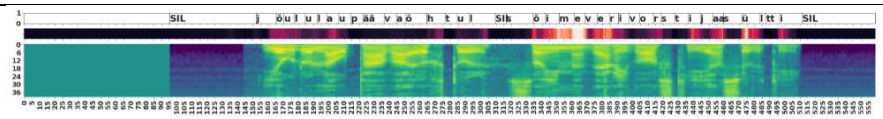
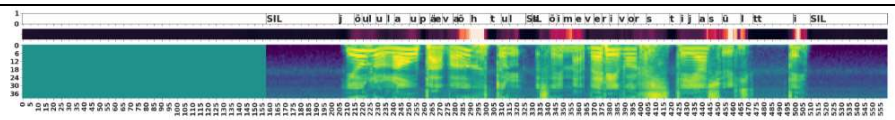
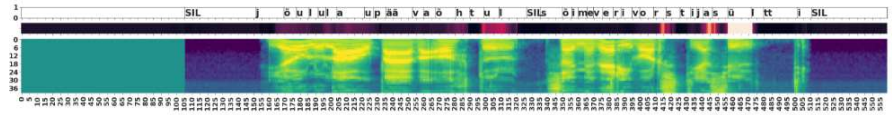
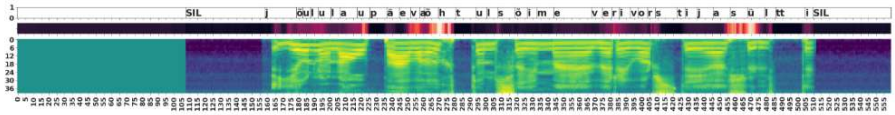
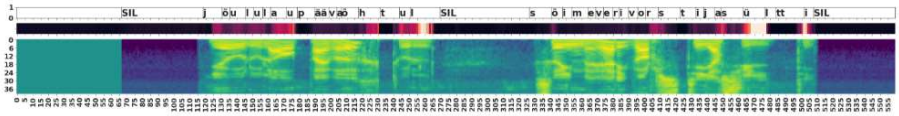
5.4 Visualization of Attention

Attention with context, when added to LSTM, enables us to visualize specific areas or timesteps that had the highest values. As described in chapter 4.1.2, we are looking for specific vowels like "*ö*" and "*õ*" or others vowels and consonants, that the attention would mark as important.

Test set contains samples that we can visualize on how different native language speakers pronounce same sentences by adding alignments extracted in previous research and the normalized importance weight α as attention, as shown in Table 7. To be more precise, we can visualize how attention mechanism has marked different timesteps importance to specific model weights. Code for visualization can be seen in Appendix 6 – Python Code for Visualizing Transcripts, Spectrogram and Attention Vector.

Table 7 contains sentence "jõulu lau päeva õhtul sõime veri vorsti ja sülti" that has been visualized with FBANK data and by pretrained best weights. As can be seen from the spectrograms and attention bar, every picked speaker has attention marked "sülti" with some or more importance. Users with *ee* and *ru* also seem to have very similar attention with current sentence regions around *ht* and *ül*.

Table 7 Sentence comparison with different speakers.

Speaker Identifier	**Accent Attention Visualization with Transcripts**
de_hk_128	
ee_mi_128	
fi_nm_128	
ru_ak_128	
fr_jo_128	

With more accuracy on attention containing models, more relevant data may be extracted on how different native language speaker's pronunciations are marked important. Comparing attention output to the real audio sound accents, marked by humans, further improvements can be made to the model and further research can be conducted on how those result can be used in future research and applications.

5.5 Experiments summary

Our goal was to achieve baseline system accuracy. Comparing different models to the goal, we can conclude that it's achievable. Convolutional architectures were not designed to be used in speech recognitions but have shown good feature extraction and generalization abilities. Additional optimization and regularization techniques are required to achieve even better results, as shown from applying extra regularizations. LSTM with attention did not achieve sufficient generalizations and further research is needed to find hyperparameters more suitable for current task.

6 Final Conclusion, Discussion and Future Work

We believe that recurrent neural network with LSTM did not achieve its full potential and further research and experiments with recurrent architectures should be conducted. Accent may need some of different approach then speech recognition. Research described in "Joint Modelling of Accents and Acoustics for Multi-Accent Speech Recognition" [22] proposed 4 layers of BILSTM with average pooling on top for utterance-level accent. Average pooling is used because speaker accent may not be observable for every timeframe but it should be for entire utterance [22]. This research could lead to better results on Estonian Foreign Accent Corpus and further experiments with LSTM is needed.

Experiments also showed that not all audio data transformations are suitable for neural networks different architectures. While MFCC does not generalise well on new data, other representations with simpler implementations, like Mel-scale filter banks, or more advanced feature extracted bottleneck features, are more suitable.

Neural network has to have lots of data to succeed in learning. EFAC corpus has a limited number of examples, approximately 80 h and this can also be a restricting factor and more data is needed to achieve better results. Using data augmentation, like generating new data or doing mix-up, can prevent models from overfitting by increasing its robustness and therefore leading to better generalization.

Other NLP solutions could use AID as part of their models to do some sort of filtering or classifications. Real world implementations that process data constantly needs fast results. This means that from a small length input, model should implement fast feature extractions and return continuous results. Current recurrent implementations using standard libraries, like Tensorflow have poor computational speeds when compared to convolutional ones. This may change in the future but it is something to take account, when selecting architectures for the real-life applications.

7 Summary

The goal of this thesis was to classify the native language of speakers based of their accents using neural networks. Accents are deviations in pronunciation that occur when speaking in a non-native language. Our objectives were to find a neural network model able to extract high-level accent specific features for classification of the native languages and to use the trained networks to determine specific regions in speech that are most important for accent identification.

It is still difficult to perform accurate accent identification or speech recognition using only raw audio and neural networks. Sound data can be transformed into different representations: filterbanks, MFCCs. Current thesis experiments have proven that different data representation have an impact on how successfully different neural network-based models can extract relevant features. While filterbanks transformations were sufficient for features extraction, MFCC failed to show good results by not generalizing well on new data. Additional transformation from MFCC features to a 40-dimensional representation with a pretrained bottleneck network, trained using Estonian speech recognition training data, achieved the best results with every model used.

Experiments made with the Estonian Foreign Accent Corpus (EFAC) proved that recurrent and convolutional neural networks are able to achieve as good or even better results, compared to baseline systems. Convolutional model with 2D filters achieved average accuracy of 82% on test dataset, while the best result of the baseline system was 66.8 % on test data.

Convolutional architectures proved to be the best models for different representation of data. They were also the fastest when training the model. Experiments made with current thesis suggest that convolutional models can be used with other than image-related tasks.

Recurrent neural network architectures are suitable for sequence processing. RNN combined with attention mechanism enables us to extract regions of the input speech that are important for accent identification. By combining those regions with the aligned speech transcripts, it is possible to determine words and phonemes that are important for native language classification.

Experiments made with different neural networks proved that they can be used with accent related tasks and an alternative to the traditional methods of phonotactic approaches and i-vectors.

Bibliography

[1] E. Meister and L. Meister, "Development and Use of the Estonian L2 Corpus," Conference: Workshop on Phonetic Learner Corpora, Satellite workshop of ICPhS, 2015.

[2] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior and K. Kavukcuoglu, "WaveNet: A Generative Model for Raw Audio," arXiv:1609.03499v2, 2016.

[3] I. Goodfellow, Y. Bengio and A. Courville, in *Deep Learning*, MIT Press, 2016.

[4] Y. Goldberg, Neural Network Methods for Natural Language Processing, Morgan & Claypool Publisher, 2017.

[5] D. J. MacKay, Information Theory, Inference, and Learning Algorithms, Version 7.2 (fourth printing), Gambridge University Press, 2005.

[6] A. Graves, A.-r. Mohamed and G. Hinton, "Speech Recognition with Deep Recurrent Neural Networks," arXiv:1303.5778v1, Toronto, 2013.

[7] V. Zocca, G. Spacagna, D. Slater and P. Roelants, Python Deep Learning, Packt Publishing, 2017.

[8] K. J. Piczak, "Environmental Sound Classification with Convolutional Neural Networks," IEEE International Workshop on Machine Learning for Signal Prpcessing, Sept. 17–20,, BOSTON, USA, 2015.

[9] B. Shi, X. Bai and C. Yao, "An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition," arXiv:1507.05717v1, 2015.

[10] G. Keren, J. Deng, J. Pohjalainen and B. Schuller, "Convolutional Neural Networks with Data Augmentation for Classifying Speakers' Native Language," 2393-2397. 10.21437/Interspeech.2016-261., 2016.

[11] F. Chollet, Deep Learning with Python, Shelter Island, NY: Manning Publications CO., 2018.

[12] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," arXiv:1412.6806v3, 2015.

[13] H. B. Soo, I. Choi and N. S. Kim, "Acoustic Scene Classification using Parallel Combination of LSTM and CNN," 2016.

[14] N. Buduma, "Fundamentals of Deep Learning," O'Reilly Media, 2017.

[15] G. Pereyra, G. Tucker, J. Chorowski, Ł. Kaiser and G. Hinton, "Regulazing Neural Networks by Penalizing Confident Output Distributions," arXiv:1701.06548v1, 2017.

[16] T. Alumäe, S. Tsakalidis and R. Schwartz, "Improved Multilingual Training of Stacked Neural Network Acoustic Models for Low Resource Languages," Proc. Interspeech 2016, 3883-3887..

[17] P. Matejka, L. Zhang, T. Ng, S. H. Mallidi, O. Glembek and B. Z. Jeff Ma, "Neural Network Bottleneck Features for Language Identification," The Speaker and Language Recognition Workshop, 2014.

[18] M. Najafian, S. Safavi, P. Weber and M. Russell, “Identification of British English regional accents using fusion of i-vector and multi-accent phonotactics ystems,” Bilbao, Spain, 2016.

[19] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola and E. Hovy, “Hierarchical Attention Networks for Document Classification,” HLT-NAACL, 2016.

[20] N. Pappas and A. Popescu-Belis, “Multilingual Hierarchical Attention Networks for Document Classification,” arXiv:1707.00896v4, 2017.

[21] F. A. R. R. Chowdhury, Q. Wang, I. L. Moreno and L. Wan, “Attention-Based Models for Text-Dependent Speaker Verification,” arXiv:1710.10470, 2017.

[22] X. Yang, K. Audhkhasi, A. Rosenberg, S. Thomas, B. Ramabhadran and M. Hasegawa-Johnson, “Joint Modelling of Accents And Acoustics for Multi-Accent Speech Recognition,” IBM T. J. Watson Research Center, 2018.

[23] B. Logan, “Mel Frequency Cepstral Coefficients for Music Modeling,” Proc. 1st Int. Symposium Music Information Retrieval, 2000.

[24] “Keras.io Documnetation,” 03 04 2018. [Online]. Available: https://keras.io/.

[25] R. Zazo, A. Lozano-Diez, J. Gonzalez-Dominguez, D. T. Toledano and J. Gonzalez-Rodriguez, “Language Identification in Short Utterances Using Long Short-Term Memory (LSTM) Recurrent Neural Networks,” PLoS ONE 11(1): e0146917. doi:10.1371/journal.pone.0146917, 2016.

[26] V. Peddinti, D. Povey and S. Khudanpur, “A time delay neural network architecture for efficient modeling of long temporal contexts.,” Sixteenth Annual Conference of the International Speech Communication Association., 2015.

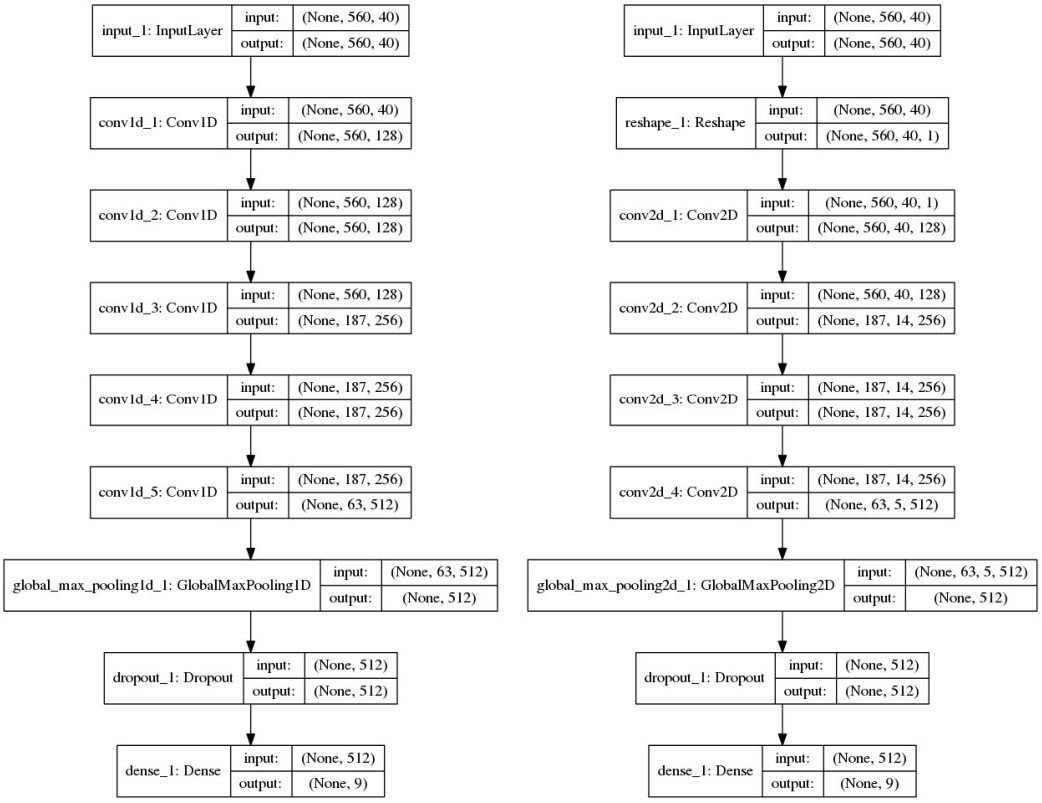
[27] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel and M. H. e. al., “The Kaldi speech recognition toolkit,” IEEE 2011 Workshop on Automatic Speech Recognition and Understanding. IEEE Signal Processing Society, 2011.

[28] M. A. Kohler and M. Kennedy, “Language identification using shifted delta cepstra,” Circuits and Systems, 2002.

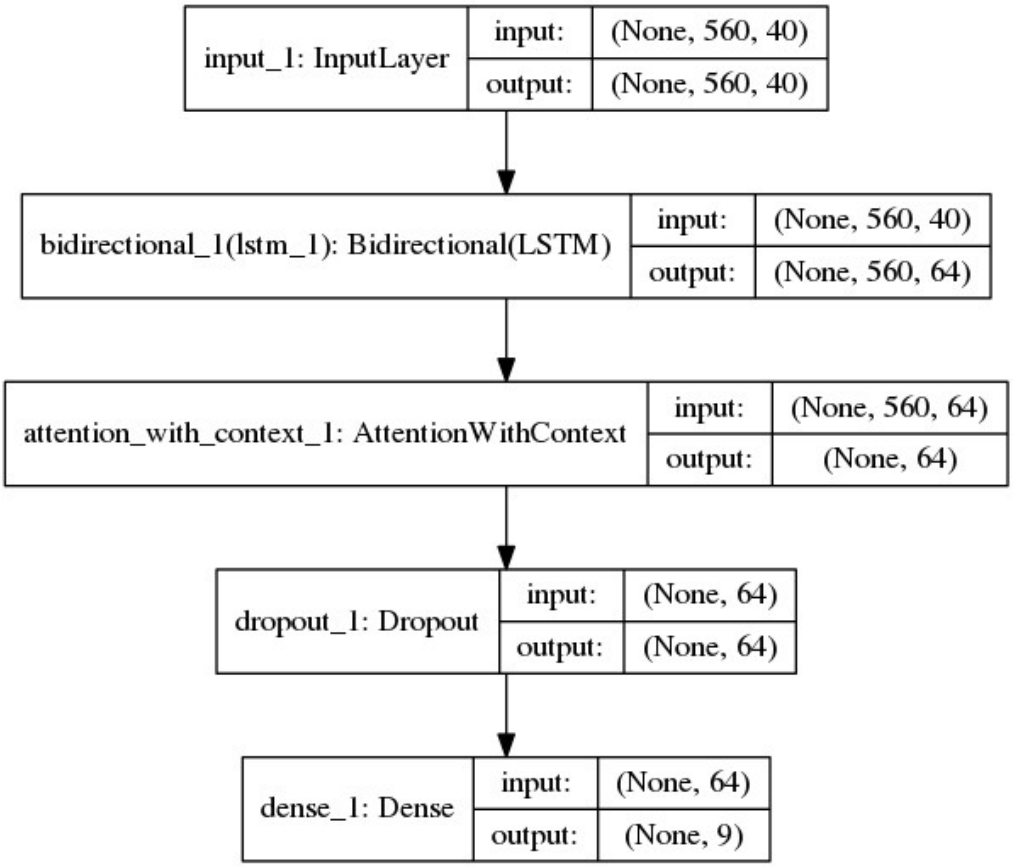
[29] L. Meister and E. Meister, “Aktsendi korpus ja võõrkeele Aktsendi uurimine,” Keel ja Kirjandus, 55(8-9), 696 - 714., 2012.

[30] “Keras Layer that implements attention mechanism,” 15 02 2018. [Online]. Available: https://gist.github.com/cbaziotis/7ef97ccf71cbc14366835198c09809d2.

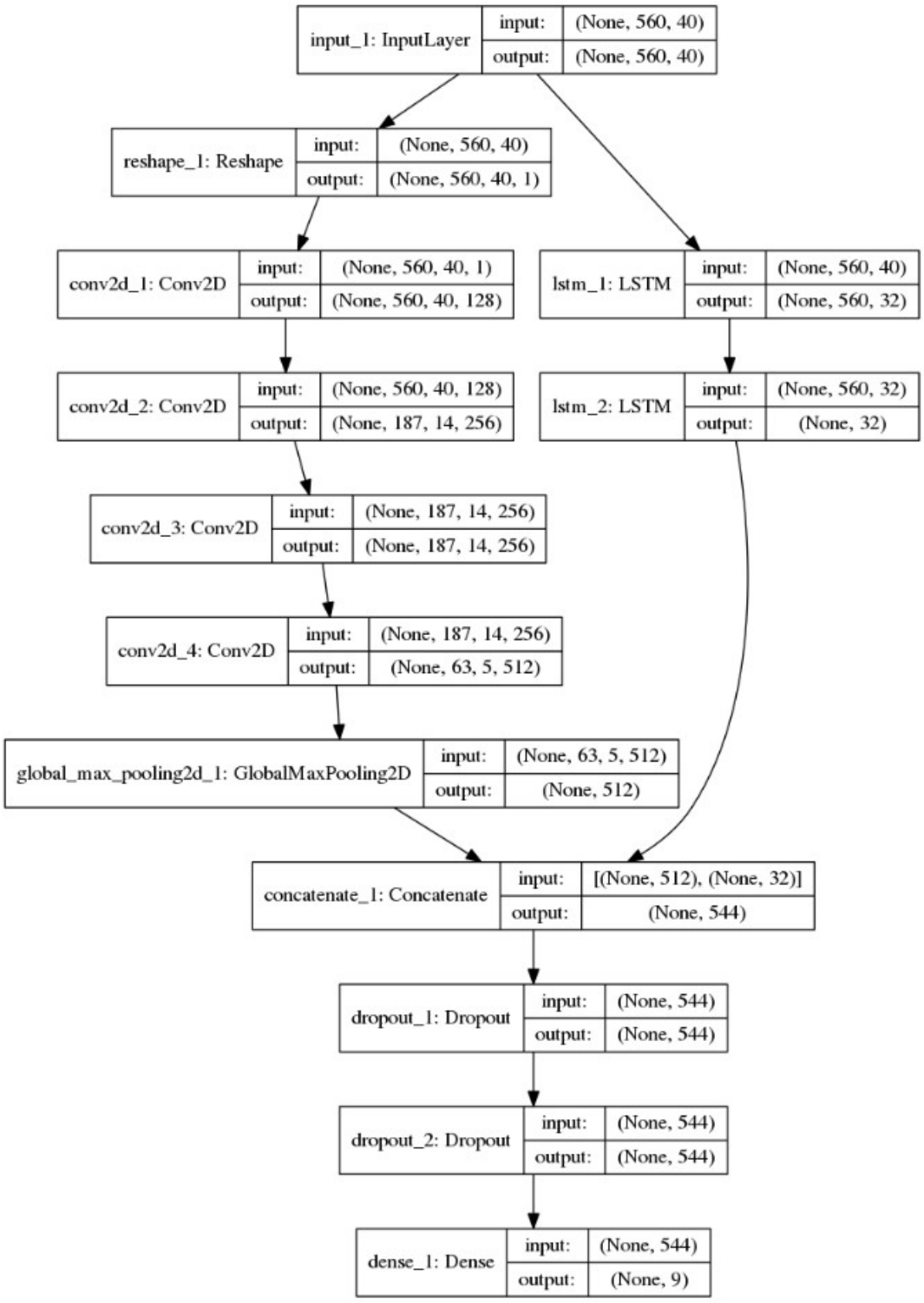
Appendix 1 – CNN with 1D and 2D in Keras



Appendix 2 – BILSTM and Attention with Context



Appendix 3 – CNN and LSTM fused model



Appendix 4 – Python Code for Training and Evaluation

```
#! /usr/bin/env python

import numpy as np
import pandas as pd
import argparse
#import collections

import time
import pickle

import glob
import os

import matplotlib.pyplot as plt
#%matplotlib inline

from keras.preprocessing import sequence
from keras.models import Model
from keras.layers import Dense, Dropout, Activation, Flatten, Layer,
Input, Reshape
from keras.layers import Conv2D, GlobalAveragePooling2D,
GlobalMaxPooling2D
from keras.utils import np_utils
from keras import backend as K
from keras import callbacks
#from keras import initializers
from keras import regularizers
#from keras import constraints
from keras import optimizers
from keras.layers import LSTM, Bidirectional
from keras.layers import concatenate
from keras.callbacks import ModelCheckpoint

import tensorflow as tf

from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
from sklearn.utils import class_weight


import itertools
```

```
#### Get input data ####
parser = argparse.ArgumentParser(description='Train and Test ESTAID
Keras Model')
parser.add_argument('train_epochs', default=5, nargs='?', help="Number
of Epochs")
parser.add_argument('train_features',  help="Train samples .npy file")
parser.add_argument('train_utt2lang',  help="Train labels file")
parser.add_argument("dev_features",  help="Dev samples .npy file")
parser.add_argument("dev_utt2lang",  help="Dev labels file")
parser.add_argument("test_features", help="Test samples .npy file")
parser.add_argument("test_utt2lang", help="Test labels file")
args = parser.parse_args()

assert int(args.train_epochs) >= 0, "Number of Epoch's value not set"

model_name = 'CONV2D'

os.makedirs(model_name, exist_ok=True)

# Hyperparameters
train_epochs = int(args.train_epochs)
maxlen = 560
batch_size = 32
kernel_regularizer = regularizers.l2(1e-3)


def load_data_estaid_tdt(features, utt2lang):
    features_data = np.load(features, encoding="bytes")
    utt2lang_data =[l.split()[1] for l in open(utt2lang)]
    return (features_data, utt2lang_data)

class TimeHistory(callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.times = []

    def on_epoch_begin(self, batch, logs={}):
        self.epoch_time_start = time.time()

    def on_epoch_end(self, batch, logs={}):
        self.times.append(time.time() - self.epoch_time_start)

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          model_name = '',
                          set_name = '',
```

```
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """

    font = {'family' : 'DejaVu Sans',
        'weight' : 'bold',
        'size'   : 22}

    plt.rc('font', **font)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        title = "Normalized confusion matrix"
        print()
    else:
        title = 'Confusion matrix, without normalization'
        print()
    print(title)
    print(cm)

    plt.figure(figsize=(15,15))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.savefig(model_name + '/' + set_name + '_confusion.png')

    plt.show()

def categorical_crossentropy_with_confidence_penalty(y_true, y_pred):
    beta = 1.0
```

```
    y_true = tf.convert_to_tensor(y_true, np.float32)
    y_pred = tf.convert_to_tensor(y_pred, np.float32)

    y_true = K.clip(y_true, K.epsilon(), 1.)
    y_pred = K.clip(y_pred, K.epsilon(), 1.)
    entropy = -K.sum(y_pred * K.log(y_pred), axis=-1)
    return K.categorical_crossentropy(y_true, y_pred) - beta * entropy


def load_data_estaid_yhat(utt2lang):
    utt2lang_data =[l.split() for l in open(utt2lang)]
    return (utt2lang_data)

print('Loading Data...')

(X_train_orig, y_train_orig) =
load_data_estaid_tdt(args.train_features, args.train_utt2lang)
print("Loaded {} training samples".format(len(X_train_orig)))
print("Loaded {} training labels".format(len(y_train_orig)))
(X_dev_orig, y_dev_orig) = load_data_estaid_tdt(args.dev_features,
args.dev_utt2lang)
print("Loaded {} training samples".format(len(X_dev_orig)))
print("Loaded {} training labels".format(len(y_dev_orig)))
(X_test_orig, y_test_orig) = load_data_estaid_tdt(args.test_features,
args.test_utt2lang)
print("Loaded {} training samples".format(len(X_test_orig)))
print("Loaded {} training labels".format(len(y_test_orig)))

y_train = y_train_orig.copy()
y_dev = y_dev_orig.copy()
y_test = y_test_orig.copy()


print('Padding Samples...')

X_train = sequence.pad_sequences(X_train_orig, maxlen=maxlen,
dtype='float')
print('X_train shape:', X_train.shape)
X_dev = sequence.pad_sequences(X_dev_orig, maxlen=maxlen,
dtype='float')
print('X_dev shape:', X_dev.shape)
X_test = sequence.pad_sequences(X_test_orig, maxlen=maxlen,
dtype='float')
print('X_test shape:', X_test.shape)


print('Encoding Labels...')
```

```
label_encoder = preprocessing.LabelEncoder()
label_encoder.fit(y_train_orig + y_dev_orig + y_test_orig)
num_classes = len(label_encoder.classes_)

y_train = np_utils.to_categorical(label_encoder.transform(y_train),
num_classes)
y_dev = np_utils.to_categorical(label_encoder.transform(y_dev),
num_classes)
y_test = np_utils.to_categorical(label_encoder.transform(y_test),
num_classes)


print('Build model...')
samples = X_train.shape[0]
timesteps = X_train.shape[1]
features = X_train.shape[2]


inputs_encoded = Input(shape=(X_train.shape[1:]))


p0 = Reshape((timesteps, features, 1))(inputs_encoded)
p1 = Conv2D(128, (3,3), strides=(1,1), activation='relu',
padding='same', kernel_regularizer=kernel_regularizer)(p0)
p1 = Conv2D(256, (5,5), strides=(3,3), activation='relu',
padding='same', kernel_regularizer=kernel_regularizer)(p1)
p1 = Conv2D(256, (5,5), strides=(1,1), activation='relu',
padding='same', kernel_regularizer=kernel_regularizer)(p1)
p1 = Conv2D(512, (5,5), strides=(3,3), activation='relu',
padding='same', kernel_regularizer=kernel_regularizer)(p1)
p2 = GlobalMaxPooling2D(data_format='channels_last')(p1)

p3 = Dropout(0.1)(p2)
results = Dense(num_classes, activation='softmax',
kernel_regularizer=regularizers.l2(1e-3))(p3)

model = Model(inputs_encoded, results)


# try using different optimizers and different optimizer configs
model.compile(
              #loss='categorical_crossentropy',
              loss=categorical_crossentropy_with_confidence_penalty,
              optimizer=optimizers.SGD(lr=0.01,momentum=0.5,
decay=0.0),
              #optimizer=optimizers.Adam(lr=0.001, decay=0.01,
amsgrad=True),
```

```
              #optimizer=optimizers.RMSprop(lr=0.001, rho=0.9,
epsilon=None, decay=0.0, clipnorm=thresh),
              #optimizer=optimizers.RMSprop(lr=0.001, rho=0.9,
epsilon=None, decay=0.0, clipvalue=1.0),
              #optimizer=optimizers.RMSprop(lr=0.001, rho=0.9,
epsilon=None, decay=0.0),
              #optimizer=optimizers.Adagrad(lr=0.01, epsilon=None,
decay=0.0),
              #optimizer=optimizers.Adadelta(lr=1.0, rho=0.95,
epsilon=None, decay=0.0),
              metrics=['accuracy'])

print("Model summary")
print(model.summary())

####### Save inital model ######
model.save(model_name + '/' + model_name + '.h5')

###### Load model ######
#model = load_model('model_name + '/model.h5',
custom_objects={'categorical_crossentropy_with_confidence_penalty':
categorical_crossentropy_with_confidence_penalty})


print('Train...')

time_callback = TimeHistory()

###### checkpoint ######
filepath = model_name + "/weights-{epoch:02d}-{val_acc:.2f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose=1,
save_best_only=False, mode='max')

class_weight_sk = class_weight.compute_class_weight('balanced',
np.unique(y_train_orig), y_train_orig)
class_weight_sk = {i:w for i,w in enumerate(class_weight_sk)}


start = time.time()

history = model.fit(X_train, y_train, batch_size=batch_size,
epochs=train_epochs, verbose=1,
          validation_data=(X_dev, y_dev),
          class_weight=class_weight_sk,
          callbacks=[ checkpoint
                    ,  time_callback])

times = time_callback.times
```

```
print('training times', times)
end = time.time()
duration = end-start
print('Toatl training time:', time.strftime("%H:%M:%S",
time.gmtime(duration)))

###### Save Trainig accuracy ######
with open(model_name + '/TrainingHistory', 'wb') as f_pi:
    pickle.dump(history.history, f_pi)
f_pi.close()

###### Save Training times ######
pickle.dump(times, open(model_name + '/' + 'TrainingTimes.pkl', "wb"))




res_files = model_name + '/*.hdf5'
result_weights = glob.glob(res_files)
max_res = 0
max_step = ''
result_weights_sorted = sorted(result_weights)
for i in result_weights_sorted:
    if int(i.split('.')[1]) >= max_res:
        max_res = int(i.split('.')[1])
        max_step =  i

print('\nMax DEV step:', max_step)


res = max_step.split('/')[-1]
if res != '':
    model.load_weights(model_name + '/'+res)
    print(model_name  + '/'+res)

print('Evaluate Dev:')
loss, acc = model.evaluate(X_dev, y_dev)
print('\nDev loss: {}, acc: {}\n'.format(loss, acc))

y_hat_dev_results = []
y_hat_test_results = []

###### Predict X_dev ##########
y_hat = model.predict(X_dev, verbose=1)
y_hat_result = label_encoder.classes_[y_hat.argmax(-1)]

###### Plot Confusion Matrix #################
```

```
cnf_matrix = confusion_matrix(y_dev_orig, y_hat_result)
np.set_printoptions(precision=2)

print(label_encoder.classes_)
plot_confusion_matrix(cnf_matrix, classes=label_encoder.classes_,
model_name = model_name, set_name='Test',
                      title='Confusion matrix, without normalization')

print('Evaluate Test:')
loss, acc = model.evaluate(X_test, y_test)
print('\nTesting loss: {}, acc: {}\n'.format(loss, acc))

###### Predict X_test ##########
y_hat_test = model.predict(X_test, verbose=1)
y_hat_test_result = label_encoder.classes_[y_hat_test.argmax(-1)]

###### Plot Confusion Matrix ########
cnf_matrix = confusion_matrix(y_test_orig, y_hat_test_result)
np.set_printoptions(precision=2)

print(label_encoder.classes_)
plot_confusion_matrix(cnf_matrix, classes=label_encoder.classes_,
model_name = model_name, set_name='Test',
                      title='Confusion matrix, without normalization')

test_yhat_save = model_name + '/' + 'yhat_results_pandas.pkl'

###### Create and save initial results to pandas datarame ######
if not glob.glob(test_yhat_save):
    (y_test_orig_yhat) = load_data_estaid_yhat(args.test_utt2lang)

pd.DataFrame(y_test_orig_yhat).set_index([0]).to_pickle(test_yhat_save
)

y_test_orig_yhat = pd.read_pickle(test_yhat_save)
y_test_orig_yhat[model_name] = y_hat_test_result
y_test_orig_yhat.to_pickle(test_yhat_save)

###### Create and save initial results to pandas datarame ######
#y_test_orig_yhat = pd.read_pickle(test_yhat_save)
#y_test_orig_yhat.head()

print('Data saved to: ' + model_name)
```

Appendix 5 – Python Code for Keras Additional Layer - Attention With Context

```
from keras import backend as K
from keras.engine.topology import Layer
from keras import initializers
from keras import regularizers
from keras import constraints

def dot_product(x, kernel):
    """
    Wrapper for dot product operation, in order to be compatible with
both
    Theano and Tensorflow
    Args:
        x (): input
        kernel (): weights
    Returns:
    """
    if K.backend() == 'tensorflow':
        return K.squeeze(K.dot(x, K.expand_dims(kernel)), axis=-1)
    else:
        return K.dot(x, kernel)


class AttentionWithContext(Layer):
    """
    Attention operation, with a context/query vector, for temporal
data.
    Supports Masking.
    Follows the work of Yang et al.
[https://www.cs.cmu.edu/~diyiy/docs/naacl16.pdf]
    "Hierarchical Attention Networks for Document Classification"
    by using a context vector to assist the attention
    # Input shape
        3D tensor with shape: `(samples, steps, features)`.
    # Output shape
        2D tensor with shape: `(samples, features)`.
    How to use:
    Just put it on top of an RNN Layer (GRU/LSTM/SimpleRNN) with
return_sequences=True.
    The dimensions are inferred based on the output shape of the RNN.
    Note: The layer has been tested with Keras 2.0.6
    Example:
        model.add(LSTM(64, return_sequences=True))
```

```
        ##model.add(AttentionWithContext())
        # next add a Dense layer (for classification/regression) or
whatever...
    """

    def __init__(self,
                 W_regularizer=None, u_regularizer=None,
b_regularizer=None,
                 W_constraint=None, u_constraint=None,
b_constraint=None,
                 bias=True, **kwargs):

        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.u_regularizer = regularizers.get(u_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.u_constraint = constraints.get(u_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        super(AttentionWithContext, self).__init__(**kwargs)

    def build(self, input_shape):
        print(input_shape)
        assert len(input_shape) == 3

        self.W = self.add_weight((input_shape[-1], input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        if self.bias:
            self.b = self.add_weight((input_shape[-1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)

        self.u = self.add_weight((input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_u'.format(self.name),
                                 regularizer=self.u_regularizer,
                                 constraint=self.u_constraint)
```

```
        super(AttentionWithContext, self).build(input_shape)

    def compute_mask(self, input, input_mask=None):
        # do not pass the mask to the next layers
        return None

    def call(self, x, mask=None):
        uit = dot_product(x, self.W)

        if self.bias:
            uit += self.b

        uit = K.tanh(uit)
        ait = dot_product(uit, self.u)

        a = K.exp(ait)

        # apply mask after the exp. will be re-normalized next
        if mask is not None:
            # Cast the mask to floatX to avoid float64 upcasting in
theano
            a *= K.cast(mask, K.floatx())

        # in some cases especially in the early stages of training the
sum may be almost zero
        # and this results in NaN's. A workaround is to add a very
small positive number epsilon to the sum.
        # a /= K.cast(K.sum(a, axis=1, keepdims=True), K.floatx())
        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(),
K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[-1]
```

Code implementing custom Keras layer for attention mechanism [30].

Appendix 6 – Python Code for Visualizing Transcripts, Spectrogram and Attention Vector

```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sbs
plt.style.use('seaborn-ticks')
from matplotlib import transforms
import matplotlib.patheffects
from matplotlib.font_manager import FontProperties

def visualize_accent(data, transcripts, attention, audio_start = 0,
vcolor = [-10, 10], save_to_file = ""):
    font = {'family' : 'DejaVu Sans',
            'weight' : 'bold',
            'size'   : 22}

    plt.rc('font', **font)

    fontp = FontProperties()
    fontp.set_size(30)
    fontp.set_weight('bold')


    fig, ax = plt.subplots(3, 1, sharex=True,
sharey=False,figsize=(50,5),gridspec_kw = {'height_ratios':[2, 2,
10]})

    # transcripts
    for l in transcripts:
        ax[0].axvline(x=float(l[0]) * 100 + audio_start,
clip_on=False, c="grey",linewidth=2)
        ax[0].text(float(l[0]) * 100 + audio_start, 0, l[2],
horizontalalignment='left', verticalalignment='bottom', fontproperties
= fontp)

    # attention
    a21 = sbs.heatmap(attention, ax=ax[1],vmin=0,
vmax=0.01,cbar=False)
    a21.set_ylabel('')
    a21.set_xlabel('')
    a21.set_yticklabels('')
    a21.set_xticklabels('')

    # spectogram
```

```
    b11 = sbs.heatmap(data.T, ax=ax[2], vmin=vcolor[0],
vmax=vcolor[1], cmap='viridis',cbar=False , center=0.)

    plt.plot()
    plt.show()

    fig.tight_layout()
    if save_to_file != '':
        filename = save_to_file+'.png'
        fig.savefig(filename)
```