BSP S3 - Speech Recognition for Luxembourgish by a Recurrent Neural Network

Friday 10th January, 2020 - 16:00

Le Minh Nguyen
University of Luxembourg
Email: le.nguyen.001@student.uni.lu

Vladimir Despotovic
University of Luxembourg
Email: vladimir.despotovic@uni.lu

***Abstract*—Luxembourgish is a West Germanic language which is mainly spoken in Luxembourg. It is spoken by roughly 390000 people worldwide. Luxembourgish remains one of Europe's under-described and under-resourced languages. In this paper, the focus will be on isolated Luxembourgish word recognition. End-to-end automatic speech recognition (ASR) is the current state-of-the-art. Deep learning replaces most modules, such as language models or acoustic models, which were used in previous ASR with a single model. To the best of our knowledge, there exist no end-to-end continuous speech recognition for the Luxembourgish language. Thus efforts were made on collecting a small audio dataset containing Luxembourgish spoken words which are speaker-dependent. We develop a speech recognition system for the Luxembourgish language based on Recurrent Neural Networks (RNNs) and compare its performance to standard feed-forward neural networks. RNNs use their internal state to operate on data time-series. Combined with connectionist temporal classification (CTC) models, they represent the current state-of-the-art in ASR. The accuracy to recognize Luxembourgish words obtained with gated RNNs, such as long short-term memory (LSTM) and gated recurrent units (GRU), were both roughly 99%.**

1. Introduction

This paper presents the Bachelor Semester Project (BSP) made by Le Minh Nguyen together with Vladimir Despotovic as his motivated tutor. The project is divided into two parts. The first part implements the classification model for spoken word recognition using the following concepts:

- Data collection and preprocessing.
- Deep Neural Networks architecture: Deep Feedforward Neural Networks and Recurrent Neural Networks, which are presented in the popular Deep Learning textbook [1].

In the second part, we explain the Deep Neural Networks architecture and compare their performance obtained for the word recognition experiment. Additionally, we investigate the reasons why gated RNNs are outperforming the baseline RNN unit.

2. Project description

2.1. Domains

- Speech Recognition
- Artificial Neural Networks
- Deep Learning
- Data preprocessing
- Training set and testing set
- Python
- Keras

2.1.1. Scientific. The scientific aspects covered by this Bachelor Semester Project are the concepts of speech recognition and Deep Learning. Different Artificial Neural Networks (ANN) architectures are investigated for the end-to-end isolated word recognition task.

Speech Recognition. The objective of speech recognition is to map an audio signal which contains a set of spoken natural language expressions to the matching sequence of words produced by the speaker. In the past, Automatic Speech Recognition (ASR) was made up of different modules such as complex feature extraction, acoustic models, language and pronunciation models. [2] Sequential models, such as Hidden Markov Models (HMM), in combination with a pre-trained language model, were used to map sequences of phones to output words. [3] A different approach is to build ASR models end-to-end. With deep learning, it replaces most of the modules with a single module. This alternative method is the main task of this report.

Artificial Neural Networks (ANN). ANNs are computing systems inspired by the biological brain. These systems are based on a set of connected units called artificial neurons. Each connection can transmit *signals* between units. A unit can process the signal and transmit it to another unit.

Deep Learning. This field deals with learning by decomposing a task's input into smaller and simpler compositions. With Deep Learning, computing systems can build complex concepts from a composition of simpler concepts.

2.1.2. Technical. The technological aspect which is covered in this project is the data collection, feature extraction and implementation of the end-to-end spoken word recognition model.

Data preprocessing. Data preprocessing is an important phase in machine learning. It ensures the quality of the gathered data by eliminating irrelevant and redundant information. Data preprocessing contains tasks such as cleaning, instance selection, normalization, feature extraction and feature selection. We will focus on feature extraction in this paper with the presentation of Mel-frequency cepstral coefficients (MFCCs) in Section 4.2.1 which are used as features extracted from the speech signal.

Training set and testing set. For a computing system to learn from and make predictions on data, a mathematical model is built from input data. This input data used to create the model consists of two datasets: The training and testing set. The training set contains pairs of an input vector, which represent features and an output vector often called the labels. With the training set, the model learns to map the input vector to the labels. On the other hand, the testing set evaluates how well the model generalizes the prediction over the dataset previously not seen by the model.

Python. This is a programming language which is interpreted, high-level and general-purpose. [4]

Keras Library. *Keras* is a high-level Neural Networks API written in Python. It is designed for easy-to-use and efficient experimentation with Deep Learning. [5]

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. One of the main deliverables is to present how to extract the features from the dataset with Mel-frequency cepstral coefficients (MFFCs). Additionally, we present the notions of Deep learning. This paper gives a brief introduction to Artificial Neural Networks (ANN) and their application for classification. Further, we extend this scientific presentation by diving deeper into two different ANNs; Feedforward Neural Networks and Recurrent Neural Networks (RNN). Special cases of RNN such as gated RNNs are furthermore investigated. This includes long short-term memory (LSTM) and gated recurrent units (GRU). Finally, we compare their performances obtained after being trained on the given audio dataset.

2.2.2. Technical deliverables. The other main deliverable for this paper is the implementation of an end-to-end speech recognition model based on the four Neural Networks, i.e. feed-forward neural networks, standard RNNs, LSTM and GRU. Additionally, we collect a small dataset of Luxembourgish spoken words $w \in \{'0', \ldots, '9'\}$ to train our model to recognize these words. To the best of our knowledge, there are no publically available datasets for speech recognition of the Luxembourgish language.

2.3. Constraints

For this BSP, we set the different constraints for the ANNs, for the speech recognition, for the Luxembourgish vocal dataset and the classification model's implementation.

Data set. First, we focus on training our model on a dataset containing English spoken numbers. [6] After having trained the model successfully on this dataset, we train it on the smaller Luxembourgish dataset and analyse its prediction accuracy. Since we aren't able to collect as much voice recordings as the English data set, we should not expect high accuracy.

Speech recognition. Since no datasets with continuous sentences of the Luxembourgish Language exists, we choose to work with isolated word recognition instead of continuous speech recognition. In isolated word recognition, words are separated by pauses.

Speaker dependency. To simplify the collection of the Luxembourgish dataset, we collect the audio recordings from one person which makes the isolated word recognition speaker-dependent.

ANNs implementation. We do not implement the Artificial Neural Networks architecture since existing deep learning libraries such as Keras are already matured. Thereby, we focus on explaining how the APIs work and how to use them efficiently in our model.

3. Pre-requisites

To start on the project, certain skills in programming and mathematics are required. In particular, the preliminary requirement of the project is as follows:

- Understanding of vector and matrix algebra.
- Introductory course in Python.
- Software development.
- Knowledge of probability and statistics is desirable although not mandatory.

3.1. Scientific pre-requisites

Linear Algebra. is a sub-field of mathematics, which works with vectors and matrices. Since the input of deep learning is data that are transformed into structures of rows and columns, linear algebra is one of the key foundations of deep learning. It is used to describe the operations of the deep learning algorithms, and implement the algorithms in code. All tasks in deep learning relate to linear algebra, from data preprocessing to the deep learning algorithms. [1]

3.2. Technical pre-requisites

Python. Python is an interpreted, high-level and general-purpose programming language, which is conceived in the late 1980s and released in 1991 by Guido van Rossum. [7] Its design philosophy accentuates readability of the code. As many high-level programming languages, Python is dynamically typed. Further, it supports multiple programming paradigms such as procedural, object-oriented and functional programming. With a proficient background in programming in Python gained during my high school and university education, I am prepared to work on this project.

Software development. Software development is the process of designing and implementation of applications and frameworks. In general, the process of software development includes writing and maintaining the source code which is often a planned and structured process. The software development contains mostly research, prototyping, modification and reuse of existing software. During the technical deliverable section, we use this structured process to design and implement our ANNs.

4. Scientific Deliverables

The scientific deliverables include; the feature extraction with MFCCs, notions of deep learning and Artificial Neural Networks (ANNs). This section concludes by a performance evaluation of four different ANNs models.

4.1. Requirements

- **FR01** Present the feature extraction with MFCCs.
- **FR02** Investigate the notions of deep learning and Artificial Neural Networks (ANNs).
  This should give an introduction to the domain of deep learning. It presents an overview of four different ANNs architectures, each of them covered by a brief introduction and theoretical aspect.
- **NFR01** Performance evaluation and comparison.
  During this section, we present and discuss the results obtained using four Neural Network architectures.

4.2. Design

4.2.1. FR01: Feature extraction with MFCCs.

Before training end-to-end ASR model, audio data have to be preprocessed. The first task is to extract features from the data which describes natural language expressions and excludes background noises and emotions.

Mel Frequency Cepstral Coefficients (MFCCs) are features widely used in ASR. MFCCs were presented by Davis and Mermelstein in the 1980s. [8]

To extract the MFCCS from the dataset, the following extraction steps have to be executed on a speech signal sampled at $16kHz$:

1. Frame the signal into $25ms$ frames. The frame length of a $16kHz$ signal:

   $$0.025 * 16000 = 400samples. \tag{1}$$

   With a frame step of 160 $samples$ or $10ms$, the frames overlap themselves, such that the first frame containing 400 $samples$ starts at sample 0 and the second frame starts at sample 160. This pattern repeats until the end of the speech signal. If the audio file is not divisible by an even number, it is padded by zeros.

2. For each frame calculate the periodogram estimate of the power spectrum. To obtain the Discrete Fourier Transform (DFT) from the frame i, we perform:

   $$S_i(k) = \sum_{N}^{n=1} s_i(n)h(n)e^{-j2\pi kn/N} \tag{2}$$

   For each speech frame $s_i(n)$, the periodogram-based power spectral estimate is calculated with:

   $$P_i(k) = \frac{1}{N} | \ S_i(k) \ |^2 \tag{3}$$

   This is called the Periodogram estimate of the power spectrum which identifies for every frame which frequencies are present.

3. Apply the Mel filterbank to the power spectrum and sum the energy in each filter. The Mel filterbank is a set of 26 triangular filters. The filterbank energies are calculated by multiplying every filterbank with the power spectrum.

4. Take the logarithm of the 26 filterbank energies.

5. Take the Discrete Cosine Transform (DCT) of the 26 log filterbank energies resulting in 26 cepstral coefficients. We only used the lower 12-13 of the 26 coefficients for ASR.

In Section 4.3.1 we will use the Python library *Librosa* to extract the MFCCs features from the dataset.

4.2.2. FR02: Deep Learning and Artificial Neural Networks.

The notions of Deep Learning and examples of ANNs presented in this section are based on the textbook *Deep Learning* [1]. These presentations should only give a high-level introduction to these notions.

4.2.3. Deep Learning.

Artificial Intelligence (AI) is a field with many practical applications, one of them being the task to understand speech. AI deals with challenging problems which are easy to perform but hard to describe formally by humans. Deep Learning is a solution to these intuitive problems. This approach to AI allows computer systems to study from experience and understand the world through a hierarchical stack of concepts. The computing system gathers knowledge from experience which eliminates the need to describe formal rules. The computer understands complex concepts with this stack of concepts by decomposing them with simpler ones. The representation of this hierarchy of concepts shows a deep graph with many layers, hence the name Deep Learning. [1]

4.2.4. Feedforward Neural Network.

Feedforward neural networks or multilayer perceptrons (MLPs) are very important deep learning models. The objective of MLPs is to approximate a given function f^*. A feedforward network defines a mapping $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the values of the parameter $\boldsymbol{\theta}$ with a given input vector $\boldsymbol{x}$ which yields the best approximation of f^*. This function can be for example a classifier, $y = f^*(\boldsymbol{x})$ which maps an input $\boldsymbol{x}$ to a class y.

These structures are called **feedforward** since there is a progress of information in one direction: From input to output.

MLPs are considered **networks** because they are made up of composed functions. Let f_1, f_2 and f_3 be three function forming a chain $f(\boldsymbol{x}) = f_3(f_2(f_1(\boldsymbol{x})))$. The chain structures are commonly used in ANNs. In this example, f_1 is called the first layer of the network, f_2 is the second layer, etc. The lenght of the chain represents the depth of the network. The final layer is also called the output layer.

During the training of the neural network, we try to find $f(\boldsymbol{x})$ to approximate $f^*(\boldsymbol{x})$. The training data contains examples of $f^*(\boldsymbol{x})$ obtained with different training points where every example $\boldsymbol{x}$ is referring to a label $y \approx f^*(\boldsymbol{x})$. These examples determine how the output layer should approximate a value close to y with each example $\boldsymbol{x}$. The other layers are called the **hidden layers** because the training examples are not determining their behaviour and it does not contain the desired output of each layer. Therefore, a learning algorithm has to determine how to utilize the hidden layers to obtain a good approximation of $f^*(\boldsymbol{x})$.

The **width** of the feedforward model is defined by the dimensionality of each hidden layer which is usually a vector containing values. These vector layers contain many **units** which process data in parallel. Each unit refers to a vector-to-scalar function, by taking input from other units and calculating its proper activation value.

We will present a simple MLP with one hidden layer which contains hidden units. The hidden layer represents a vector of hidden units $\boldsymbol{h}$ which are calculated by a function $f_1(\boldsymbol{x}; \boldsymbol{\theta})$. $\boldsymbol{\theta}$ consists of $\boldsymbol{W}$ and $\boldsymbol{b}$ which are the $weights$ and $biases$ repectively such that:

$$\begin{aligned} h &= f_1(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) \\ \Leftrightarrow h &= \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b} \end{aligned} \tag{4}$$

$\boldsymbol{h}$ is then used as input for the second layer which is the output layer; $f_2(\boldsymbol{h}; \boldsymbol{V}, \boldsymbol{c})$, with V being the weigth matrix and c the bias vector.

$$\begin{aligned} y &= f_2(\boldsymbol{h}; \boldsymbol{V}, \boldsymbol{c}) \\ \Leftrightarrow y &= \boldsymbol{V}^T \boldsymbol{h} + \boldsymbol{c} \end{aligned} \tag{5}$$

The network can be illustrated as a chain of two functions:

$$\begin{aligned} & f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}) \\ & \quad = f_2(f_1(\boldsymbol{x})) \end{aligned} \tag{6}$$

After using an affine transformation, the hidden units are computed by a nonlinear function g called an activation function:

$$h = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}) \tag{7}$$

The common activation function in deep learning is the **rectified linear unit** or ReLU defined as:

$$g(z) = \max\{0, z\} \tag{8}$$

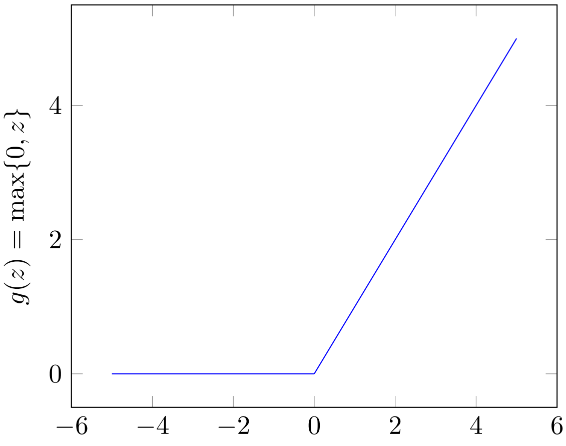


Figure 1: The rectified linear activation function.

The complete feedforward network looks as follows:

$$\begin{aligned} & f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}) \\ & = \boldsymbol{V}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}\} + \boldsymbol{c} \end{aligned} \tag{9}$$

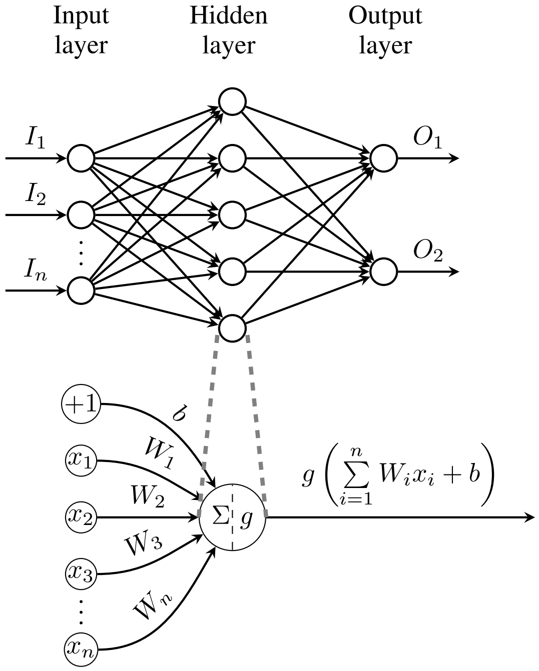


Figure 2: A figure of a single perceptron with its position in a large-scale MLP.

A representation of a single perceptron and its position in an MLP is given in Figure 2.

Gradient-Based Learning. During the training of the model, deep learning algorithms solve an optimization problem. The problem is to minimize the error represented by an objective function (loss). With the loss function the deep learning model optimizes the approximation of f^*. An example of a loss function for MLP can be the mean squared error (MSE):

$$\frac{1}{n}\sum_{i=1}^{n}(f^*(\boldsymbol{x}_i) - f(\boldsymbol{x}_i;\boldsymbol{\theta}))^2 \tag{10}$$

ANNs usually uses gradient-based optimizers to obtain a minimal loss function.

Back-propagation. Gradient descent is a back-propagation algorithm which performs a backward pass through the network while adjusting the model's parameters which are the weights and biases. It repeatedly modifies the parameters of the units in the network to minimize the error between the predicted and desired output vector. The algorithm passes information from the loss function backwards through the network. It computes the gradient while passing back the network. There are three types of gradient descent, namely batch gradient descent, stochastic gradient descent (SGD) and mini-batch gradient descent. Compared to batch gradient descent, SGD selects randomly a data sample instead of a whole data batch for each backward iteration.

4.2.5. Recurrent Neural Network.

Recurrent Neural Networks (RNNs) are a particular architecture of ANNs which can process sequential data $S = x_1, \ldots, x_n$. In traditional ANN the input and output data are assumed to be independent. However, for sequential data, this is usually not the case. For example, the prediction of the next word in a sentence is dependent on the words that appeared before. As opposed to MLP, RNN feeds back their outputs back to their network and thus gain an overview of past inputs. To explain how RNNs are implemented, we have to introduce the notion of computational graphs. A computational graph is a formal structure representing a set of computations. We obtain a chain of events by unfolding a recurrent computation into a computational graph with a repetitive structure.

Consider a recurrent system,

$$s_t = f(s_{t-1};\theta) \tag{11}$$

with s being the state of the system. The graph can be unfolded considering the time step $t = 3$, we have,

$$\begin{aligned} s_3 &= f(s_2;\theta) \\ &= f(f(s_1;\theta);\theta) \end{aligned} \tag{12}$$

This expression can be illustrated by a directed acyclic computational graph, as shown in Figure 3

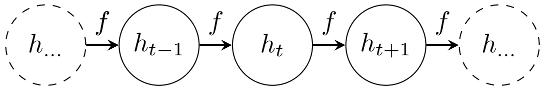


Figure 3: The recurrent system described by equation 12, illustrated as an unfolded computational graph.

Now we consider the recurrent system accepting an external signal $\boldsymbol{x}_t$,

$$s^t = f(s^{t-1}, \boldsymbol{x}_t;\theta) \tag{13}$$

where the state s is containing information about the past sequence S. The hidden units can be described with equation 13 using $\boldsymbol{h}$ to denote the state,

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t;\theta) \tag{14}$$

represented in Figure 4.

With graph unfolding, we can now present some common examples of an RNN. Here are some important designs of RNNs:

- RNNs which take a single input and produce an output at every time step t. This is called a one-to-many RNN.

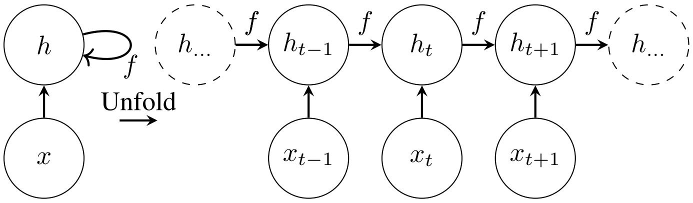


Figure 4: RNN with no outputs, it processes the input x and passes it into the state h which is then passed through time. On the right graph, the same network is unfolded as a computational graph. [1]

- RNNs which read an entire data sequence and produce a data sequence. This is called a many-to-many RNN.
- RNNs which are connected recurrently between hidden units, read an entire data sequence S and produce only one single output. This is called a many-to-one RNN.

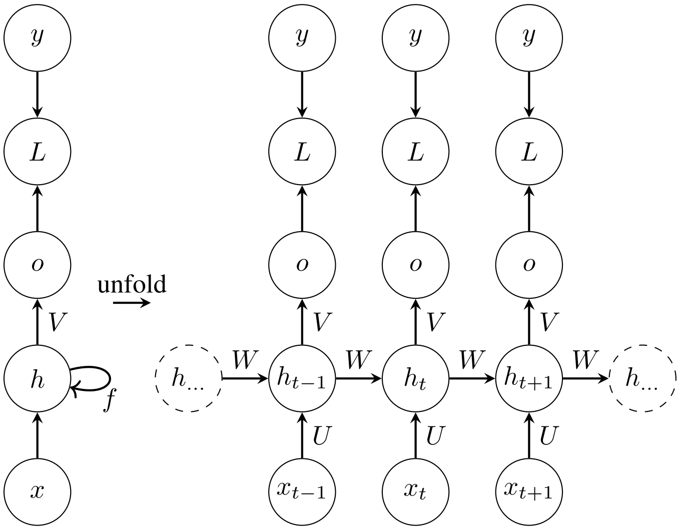


Figure 5: This computational graph is for calculating the training loss of an RNN which labels a sequence of input values x to corresponding output values o. A loss function L is calculating the distance from o to the corresponding target y. On the left, the network is represented as a recurrent graph. On the right, the same network is unfolded as a computational graph. [1]

We pick the RNN represented in Figure 5 to develop forward propagation equations. This RNN takes in a data sequence S and outputs for every time step t an output. In this figure, we didn't include an activation function for the hidden units h. However, for the equations, we assume the activation function $\mathcal{H}$ to be the hyperbolic tangent.

Let h_0 be the initial state. For every time step from $t = 1$ to $t = n$, we apply the following equations:

$$\boldsymbol{a}_t = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t \tag{15}$$
$$\boldsymbol{h}_t = \tanh(\boldsymbol{a}_t) \tag{16}$$
$$\boldsymbol{o}_t = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}_t \tag{17}$$
$$\hat{\boldsymbol{y}}_t = softmax(\boldsymbol{o}_t) \tag{18}$$

where $\boldsymbol{b}$ and $\boldsymbol{c}$ are the bias vectors and $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ are the weight matrices.

The Challenge of Long-Term Dependencies. The challenge of learning long-term dependencies is that gradients, propagated through a very deep RNN, can vanish or explode. Gradients can vanish when the gradients are converging to 0 and therefore the weights won't be updated while learning. Exploding gradients happen if the gradients are too large and the loss function will never reach the optimized minimum. As a solution to the challenges of RNNs, effective sequence models used in practice are called gated RNNs. These include the long short-term memory (LSTM) and the gated recurrent unit (GRU) based RNNs. Gated RNNs are based on the idea of creating paths through time that have gradients that neither vanish nor explode. [9]

4.2.6. Long short-term memory.

An LSTM unit is composed of a memory cell, an input gate, an output gate and a forget gate. The cell stores information over a long period and the three gates control the flow of information through the cell. RNNs using LSTM units deal with the vanishing gradient problem because these units allow gradients to flow unchanged. [9]

The standard formulation of a single LSTM cell can be given by the following equations:

$$f_t = \sigma(W_f h_{t-1} + V_f x_t + b_f) \tag{19}$$
$$i_t = \sigma(W_i h_{t-1} + V_i x_t + b_i) \tag{20}$$
$$C_t^{'} = \tanh(W_C h_{t-1} + V_C x_t + b_C) \tag{21}$$
$$C_t = f_t C_{t-1} + i_t C_t^{'} \tag{22}$$
$$o_t = \sigma(W_o h_{t-1} + V_o x_t + b_o) \tag{23}$$
$$h_t = o_t \tanh(C_t) \tag{24}$$

where σ is the sigmoid function, $\tanh$ is the hyperbolic tangent function, $f, i, C^{'}, C, o$ are the forget gate, input gate, new memory cell content, and memory cell content, output gate respectively. See representation of an LSTM cell in Figure 6.

4.2.7. Gated Recurrent Unit.

Identical to LSTM units, the gated recurrent unit (GRU) consists of gated units which control the flow of information through the unit, although without having a

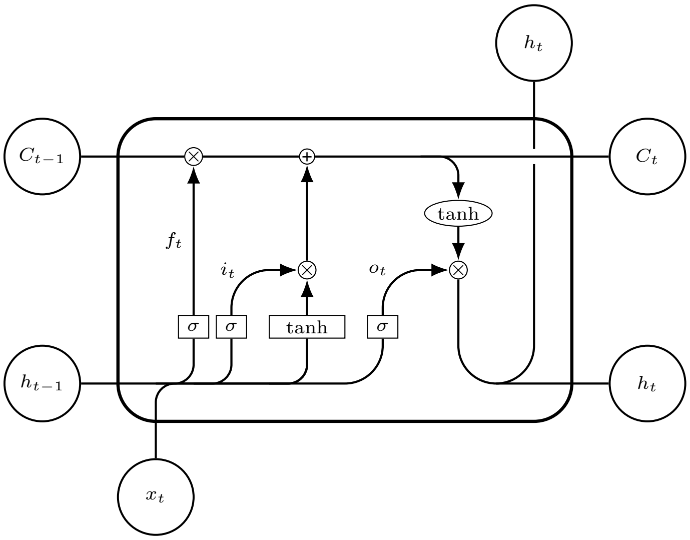


Figure 6: LSTM unit

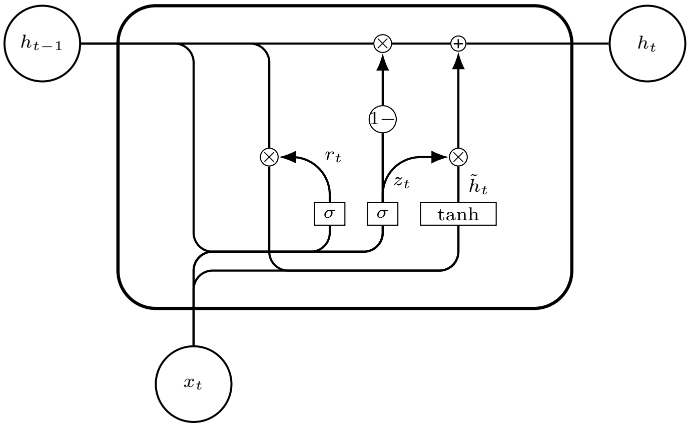


Figure 7: GRU unit.

dedicated memory cell. Compared to LSTM units, GRUs don't have an output gate. The output gate controls the amount of memory content seen by the other units in the network. Thus GRU exposes its entire content without control. [10]

The activation of a GRU unit is computed with:

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t \tag{25}$$

with an update gate z_t which decides how much the unit updates its content. Its content consists of the previous activation h_{t-1} and the candidate activation $\tilde{h}_t$. The update gate is given by the equation:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{26}$$

This update gate is similar to the one used by LSTM. However, it exposes its state each time step since it does not have a procedure to control how the state is exposed. The candidate activation is given by:

$$\tilde{h}_t = \tanh(Wx_t + U(r_t \odot h_{t-1})) \tag{27}$$

where r_t is a reset gate and is computed comparably to the update gate:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{28}$$

See the illustration of a GRU unit in Figure 7.

4.3. Production

4.3.1. Feature extraction with MFCC in Python.

For the pilot study, we use an audio dataset of English spoken words [6]. It contains 105829 utterances of 35 words. This dataset contains 2618 persons pronouncing these different words. Every utterance is roughly $1s$ long, sampled at $16kHz$ and saved as a WAVE format file. From this dataset we choose the words $w \in \{'zero', 'one', 'two', \ldots, 'eight', 'nine'\}$.

We use the Python library *librosa* [11] to extract the MFCCs from the dataset. Instead of processing the whole dataset, the following steps are applied to one single audio file f.

1. Let f be an audio file sampled at $16kHz$, *Librosa* loads the song as follows:

```
import librosa

y, sr = librosa.load(f, sr=16000, mono=True,
    duration=1)
```

   with y being the audio time series and sr the sample rate of y.

2. The next step is to pad the time series y if it is smaller than $1s$:

```
import numpy as np

if y.size < 16000:
  rest = 16000 - y.size
  left = rest // 2
  right = rest - left
  y = np.pad(y, (left, right), 'reflect'))
```

   The time series is reflected on both sides to get the correct size.

3. After matching the time series to the correct size, the MFCCs can be extracted with:

```
mfccs = librosa.feature.mfcc(y=y, sr=sr)
```

   The function *.mfcc()* returns an *np.ndarray* with the dimension:

$$shape = (n\_mfcc, t)$$

   with n_mfcc being the number of MFCCs and t the number of frames.

4. The last step is to flatten this matrix to a vector which can be stored in a data frame for later training.

```
row = mfccs.T.flatten()
```

The matrix $mfccs$ is transposed and flattened. This concatenates the columns together forming a vector.

These steps are then applied to the entire dataset. Each computed vector is appended to the data frame which is going to be stored as a *.csv* file. The entire code snippet is given in the Appendix in Figure 26.

4.4. Assessment

4.4.1. NFR01: Performance evaluation.

In this section, the performance of the four ANNs investigated in Section 4.2.2 are evaluated. We use the English and Luxembourgish dataset for the performance analysis.

The English and Luxembourgish dataset contains spoken words $w \in \{'0', \ldots, '9'\}$. The English dataset consists of 17749 data instances whereas the Luxembourgish consists of 500 data instances.

The first step is to choose an optimal optimizer. The choice is between using the stochastic gradient descent (SGD) or Adam optimizer. The latter is a combination of two extensions of SGD, Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp) [12]. The four models will be trained on the English dataset to designate the right optimizer and number of epochs. Their training will be visualized with 60 *epochs* each. The training visualization is given in the appendix as follows:

1. Feedforward Neural Network:
   - Validation accuracy in Figure 8.
   - Validation loss in Figure 9.
2. RNN:
   - Validation accuracy in Figure 10.
   - Validation loss in Figure 11.
3. LSTM:
   - Validation accuracy in Figure 14.
   - Validation loss in Figure 15.
4. GRU:
   - Validation accuracy in Figure 16.
   - Validation loss in Figure 17.

Let $n = 60$ be the number of epochs. The two optimizers return the following testing accuracies (acc) and losses:

Optimizer	FF		RNN		LSTM		GRU	
	acc	loss	acc	loss	acc	loss	acc	loss
SGD	0.79	0.65	0.43	1.51	0.86	0.46	0.85	0.46
Adam	0.87	0.47	0.72	0.97	0.94	0.24	0.94	0.24

TABLE 1: Comparison of the four models' testing results using SGD and Adam. The models were trained on the English dataset.

Table 1 shows that the validation accuracies using the Adam optimizer yield better accuracies compared to SGD. Therefore, for the following investigation, the Adam optimizer is considered.

The next step is to choose the number of epochs which will be used for the performance analysis. The training visualizations show the loss converges around 40 epochs except for the RNN model. Figure 12 shows that the RNN model converges around 220 epochs with an accuracy of 0.88. The training visualization of the RNN model with 300 epochs is given in Figures 12 and 13 in the appendix.

Choosing $epochs = 40$ to evaluate each model's performance, we have:

Optimizer	FF		RNN		LSTM		GRU	
	acc	loss	acc	loss	acc	loss	acc	loss
Adam	0.83	0.54	0.63	1.10	0.937	0.23	0.94	0.23

TABLE 2: The testing results of the four models using Adam. The four models were trained on the English dataset. FF is the abbreviation for feedforward and acc stands for accuracy.

Inspecting the results shows that LSTM and GRU perform better than feedforward and RNN. The overall performance is slightly underneath the performance at 60 epochs. In this case, RNN has the lowest accuracy. This is expected, since standard RNNs can't deal with learning long-term temporal dependencies, because the gradient of the loss function decays exponentially with time (vanishing gradient problem). LSTMs and GRUs use gated units to maintain the information in memory for longer periods of time, and to control which information is kept, and which is forgotten. This explains substantially better performance obtained using LSTM and GRU. Compared to feedforward neural networks, LSTM and GRU have better accuracy since they take advantage of their recurrent units to process series data.

Training on Luxembourgish dataset. The same models used to train on the English dataset were used to train on the Luxembourgish dataset. The following testing accuracies and losses were obtained while training on the Luxembourgish dataset:

Optimizer	FF		RNN		LSTM		GRU	
	acc	loss	acc	loss	acc	loss	acc	loss
Adam	0.84	0.59	0.73	0.83	1.0	0.02	0.99	0.02

TABLE 3: The testing results of the four models using Adam. The four models were trained on the Luxembourgish dataset. The dataset contains 50 instances of each spoken word. FF is the abbreviation for feedforward and *acc* stands for accuracy.

The results show that LSTM and GRU generalize better than feedforward and RNN. The validation accuracies of LSTM and GRU are both roughly 99%. The high accuracy of LSTM and GRU can be explained by the small dataset which is also speaker-dependent. The results of LSTM and GRU are expected to drop if the dataset grows and contains a variety of speakers.

The training visualization on the Luxembourgish dataset is given in the appendix as follows:

1. Feedforward Neural Network:
   - Validation accuracy in Figure 18.
   - Validation loss in Figure 19.
2. RNN:
   - Validation accuracy in Figure 20.
   - Validation loss in Figure 21.
3. LSTM:
   - Validation accuracy in Figure 22.
   - Validation loss in Figure 23.
4. GRU:
   - Validation accuracy in Figure 24.
   - Validation loss in Figure 25.

5. Technical Deliverables

As technical deliverables, the four ANN models are implemented in Python using the Keras [5] library and a small dataset containing Luxembourgish spoken words is collected.

5.1. Requirements

- **FR01** Implementation of the four classification models.
  We use the Keras library to implement our models with four different Neural Network architecture.
- **FR02** Collect a small dataset of Luxembourgish spoken words.
  We collect a small dataset containing Luxembourgish audio samples containing the words $w \in \{'0', \dots, '9'\}$.

5.2. Design

Keras is a high-level Neural Networks API written in Python. It is designed for easy-to-use and efficient experimentation with Deep Learning. [5]

The main data structure in Keras is a model. The common model is the *Sequential* model which organizes layers in a linear stack. An example of a *Sequential* model in Python is given below:

```
from keras.models import Sequential

model = Sequential()
```

To stack layers in the model, the *.add()* function is used:

```
from keras.layers import Dense

model.add(Dense(unit=64, activation='relu',
                input_dim=100))
model.add(Dense(unit=10, activation='softmax'))
```

After stacking up the model, its learning process has to be configured with *.compile()* function:

```
modl.compile(loss='categorical_crossentropy',
             optimizer='sgd',
             metrics=['accuracy'])
```

Now the training data can be iterated in batches:

```
model.fit(x_train, y_train, epochs=5, batch_size
    =32)
```

To evaluate the performance of the model, the *.evaluate()* function is called:

```
loss_and_metrics = model.evaluate(x_test, y_test,
    batch_size=128)
```

5.3. Production

5.3.1. Implementation of classification models.

In this section, the implementation of the four ANN models are presented.

First of all, we initialize the training and testing sets:

```
import numpy as np
from sklearn.model_selection import
    train_test_split

df = pd.read_csv('largetrainingset.csv')
y = df['label']
X = df.drop(columns=['label'])

X_train, X_test, y_train, y_test =
    train_test_split(X,y)
```

1. Feedforward neural networks:

```
feedForward = Sequential()
feedForward.add(Dense(256, activation='relu',
    input_shape=(X_train.shape[1],)))
feedForward.add(Dropout(0.2))
feedForward.add(Dense(128, activation='relu'))
feedForward.add(Dropout(0.2))
feedForward.add(Dense(64, activation='relu'))
feedForward.add(Dropout(0.2))
feedForward.add(Dense(10, activation='softmax'
    ))
feedForward.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

```

historyfeedForward = feedForward.fit(X_train,
    y_train, validation_split=0.25,
    epochs=60, batch_size=128, verbose=1)
test_loss, test_acc_ff = feedForward.evaluate(
    X_test, y_test)
```

The sequential model is used to stack four *Dense* layers, which are fully connected layers, meaning that all neurons in one layer are connected to all neurons in the subsequent layer. This model is composed of one input layer, two hidden layers and one output layer. The first dense layer takes as input a vector with the same dimensions as a row in the dataset. The first three layers are using *ReLU* as activation function whereas the output layer is using *softmax*, which is a standard choice for multi-class classification problems. In between the dense layers, dropout layers are stacked. These dropout layers are dropping units from the model to prevent overfitting. The parameter *loss* of the *.compile()* function is set to *'sparse_categorical_crossentropy'* since the output should be an integer tensor instead of a binary vector.

Before continuing with the recurrent models, the input data has to be reshaped since RNN, LSTM and GRU are taking in 3D shaped input data:

```
X_train = np.reshape(X_train.values,(X_train.
    shape[0],32,20))
X_train, X_test, y_train, y_test =
    train_test_split(X_train,y)
```

2. RNN:

```
rnn = Sequential()
rnn.add(SimpleRNN(256, return_sequences=True,
                  input_shape=(X_train.shape
    [1],X_train.shape[2])))
rnn.add(Dropout(0.3))
rnn.add(SimpleRNN(128))
rnn.add(Dropout(0.3))
rnn.add(Dense(10, activation='softmax'))
rnn.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy',
                metrics=['accuracy'])

historyrnn = rnn.fit(X_train, y_train,
    validation_split=0.25, epochs=50,
                      batch_size=128, verbose
    =1)
test_loss, test_acc_rnn = rnn.evaluate(X_test,
     y_test)
```

The RNN model is implemented in the same manner as the feedforward model. Although, this model is only stacked with one hidden layer. Instead of using dense layers, this model uses *SimpleRNN* layers. The parameter *return_sequences* is set to *True* so that the next simpleRNN layer has access to all the hidden states from the previous layer. The output layer is again a dense layer with *softmax* as the activation function. As input, it takes in a matrix, which is an instance of the 3D dataset.

3. LSTM:

```
lstm = Sequential()
lstm.add(LSTM(256, return_sequences=True,
                  input_shape=(X_train.shape
    [1],X_train.shape[2])))
lstm.add(Dropout(0.3))
lstm.add(LSTM(128))
lstm.add(Dropout(0.3))
lstm.add(Dense(10, activation='softmax'))
lstm.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy',
                 metrics=['accuracy'])
historylstm = lstm.fit(X_train, y_train,
    validation_split=0.25, epochs=50,
                      batch_size=128, verbose
    =1)
test_loss, test_acc_lstm = lstm.evaluate(
    X_test, y_test)
```

The LSTM model uses *LSTM* layers and is composed of one hidden layer as well. Its parameter *return_sequences* is set to *True* with the same reason as for the RNN model. The LSTM model has the same output layer as the two previous ones.

4. GRU:

```
gru = Sequential()
gru.add(GRU(256, return_sequences=True,
                  input_shape=(X_train.shape
    [1],X_train.shape[2])))
feedForward.add(Dropout(0.3))
gru.add(GRU(128))
feedForward.add(Dropout(0.3))
gru.add(Dense(10, activation='softmax'))
gru.compile(optimizer='adam', loss='
    sparse_categorical_crossentropy',
                 metrics=['accuracy'])

historygru = gru.fit(X_train, y_train,
    validation_split=0.25, epochs=50,
                    batch_size=128, verbose
    =1)
test_loss, test_acc_gru = gru.evaluate(X_test,
     y_test)
```

The GRU model uses *GRU* layers and is composed of one hidden layer. Its parameters are set to the same values as the RNN and LSTM model. It uses the same dense layer activated by the *softmax* function as output layer.

5.3.2. FR02: Luxembourgish dataset collection.

To collect the Luxembourgish dataset, the *Teachable Machine* [13] was used. It contains a recorder to create a dataset consisting of audio files. The recording of the audio files can be set to $1s$. However, the recorder saves the files in the *.webm* file type. Therefore, the audio files have to be converted to *.wav* files before the features can be extracted. The following bash script using the *ffmpeg* command converts the audio files in a file directory to the desired file format:

```
#!/bin/zsh

for folders in letzdatasetraw/*
do
  for file in $folders/*
  do
    if [[ ${file: -5} == ".webm" ]]
    then
      ffmpeg -i $file letzdatasetwav/${folders##
    */}/${${file##*/}%.*}_$(date +%Y%m%d_%H%M%S).
    wav
    fi
  done
done
```

5.4. Assessment

All the technical requirements were accomplished. The basic usage of the *Keras* API was presented. With this, the implementation of the four ANNs in Keras was explained in Section 5.3.1. Additionally, the methods for collecting the Luxembourgish dataset were shown.

6. Conclusion

In this paper, end-to-end isolated word recognition using deep neural networks is presented. Four different ANN models were investigated. The focus of this paper is using RNN architectures for end-to-end speech recognition and investigating whether gated RNNs outperform vanilla RNN units and how they deal with long-term dependencies. The results showed, that gated RNNs substantially outperform basic feedforward and recurrent neural networks. The performance analysis can be consulted in Section 4.4.1

A further objective of this paper is to recognize Luxembourgish words. To the best of our knowledge, there is a lack of continuous speech dataset in Luxembourgish. Thus efforts were put to collect a small dataset containing Luxembourgish spoken words $w \in \{'Null', \ldots, 'Neng'\}$. This dataset was recorded by one person which makes it speaker-dependent speech recognition. In Section 5.3.2 the methods used to collect the dataset were elaborated. Four previously discussed ANN architectures were applied to our collected Luxembourgish dataset and yielded excellent performance with accuracy close to 1.

Continuation of this project would be to extend the Luxembourgish dataset with continuous speech or to add instances spoken by multiple speakers.

In my opinion, I find it interesting to work with deep neural networks and their practical use. This project will give me the required background to start on my next BSP which deals with attention-based models for speech recognition.

Acknowledgment

I would like to thank my tutor Vladimir Despotovic for his constructive feedback and mentorship. His introduction and explanation of neural networks were outstanding. Additionally, I thank him for supervising my paper. I would recommend fellow BiCS Students interested in this field to work with Vladimir Despotovic.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. H. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015. [Online]. Available: http://arxiv.org/abs/1512.02595

[3] W. S. J. Cai, "End-to-end deep neural network for automaticspeech recognition," 2015. [Online]. Available: https://cs224d.stanford.edu/reports/SongWilliam.pdf

[4] "Python software foundation, python language reference, version 3.7. available at," http://www.python.org/.

[5] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[6] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *CoRR*, vol. abs/1804.03209, 2018. [Online]. Available: http://arxiv.org/abs/1804.03209

[7] "General python faq," https://docs.python.org/3/faq/general.html#what-is-python, accessed 19/05/19.

[8] "Mel frequency cepstral coefficient (mfcc) tutorial," http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/#references, accessed 02/12/19.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[10] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

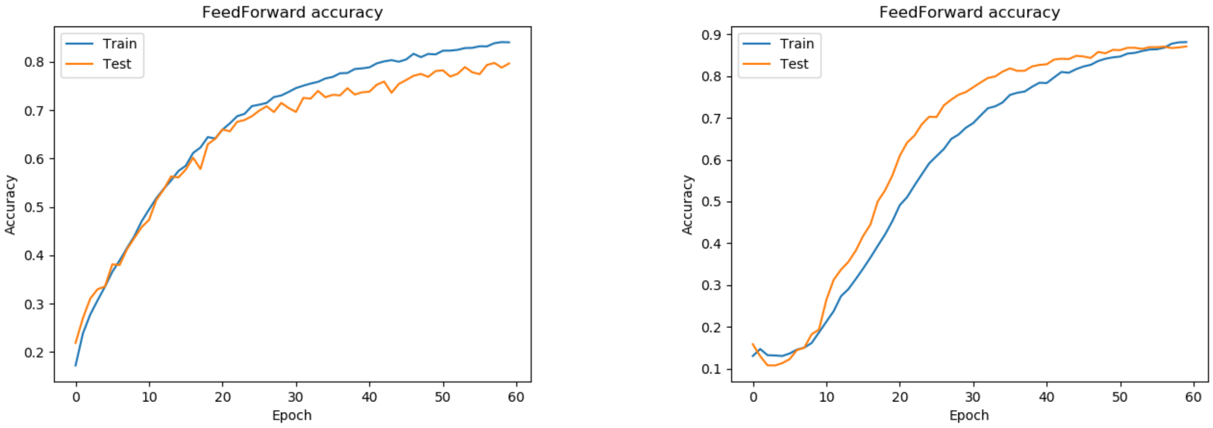
[11] [Online]. Available: http://doi.org/10.5281/zenodo.3478579

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[13] "Teachable machine," https://teachablemachine.withgoogle.com/, accessed: 23/12/19.

7. Appendix

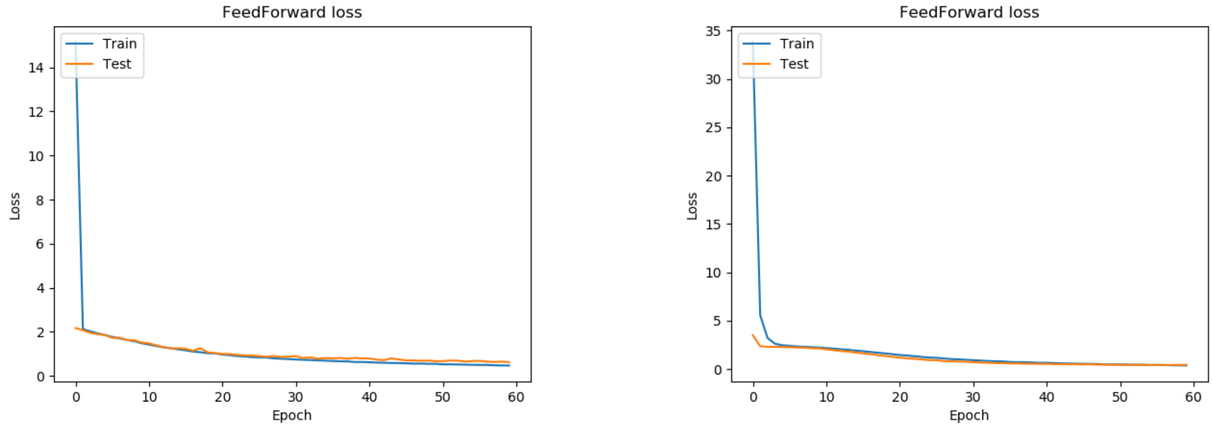
Training on English dataset:



(a) Feedforward's accuracy using SGD.

(b) Feedforward's accuracy using Adam.

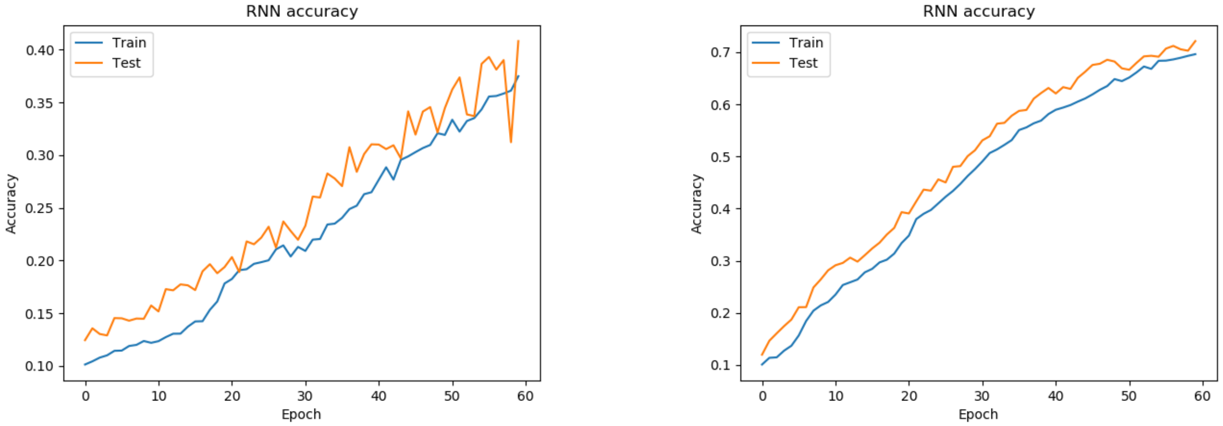
Figure 8: Accuracy comparison between feedforward using SGD and Adam.



(a) Feedforward's loss using SGD.

(b) Feedforward's loss using Adam.

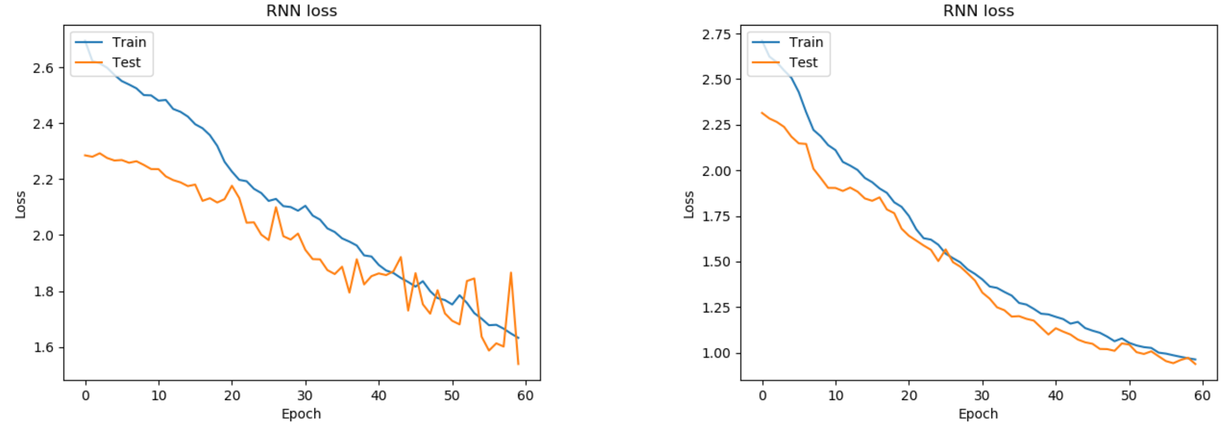
Figure 9: Loss comparison between feedforward using SGD and Adam.



(a) RNN's accuracy using SGD.

(b) RNN's accuracy using Adam.

Figure 10: Accuracy comparison between RNN using SGD and Adam.



(a) RNN's loss using SGD.

(b) RNN's loss using Adam.

Figure 11: Loss comparison between RNN using SGD and Adam.

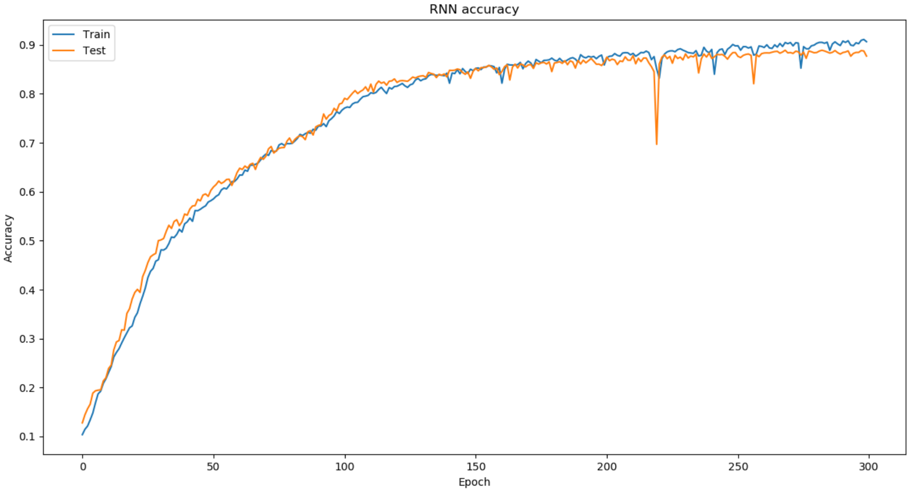


Figure 12: Training and validation accuracy of RNN model.

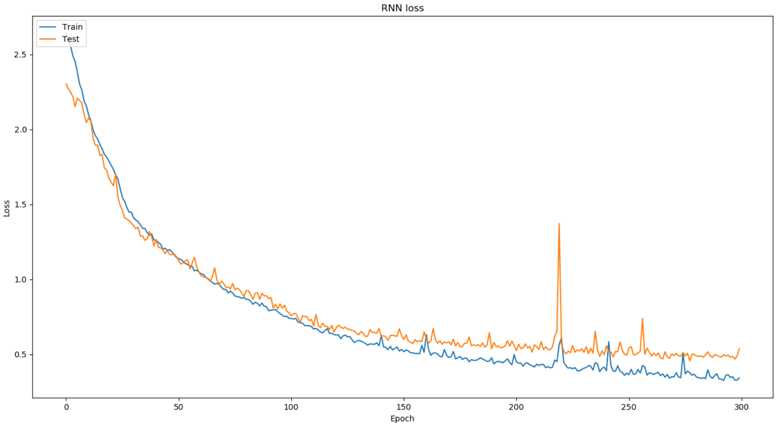
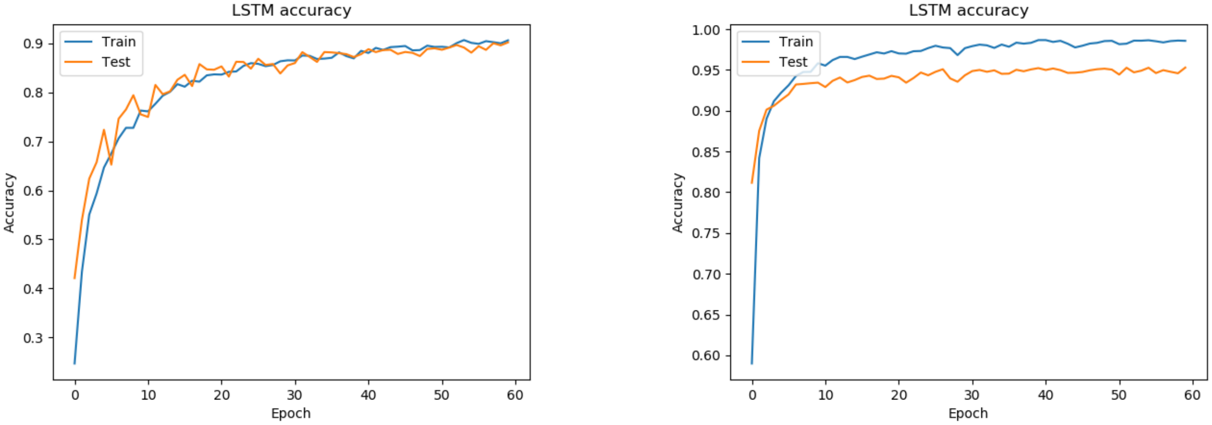


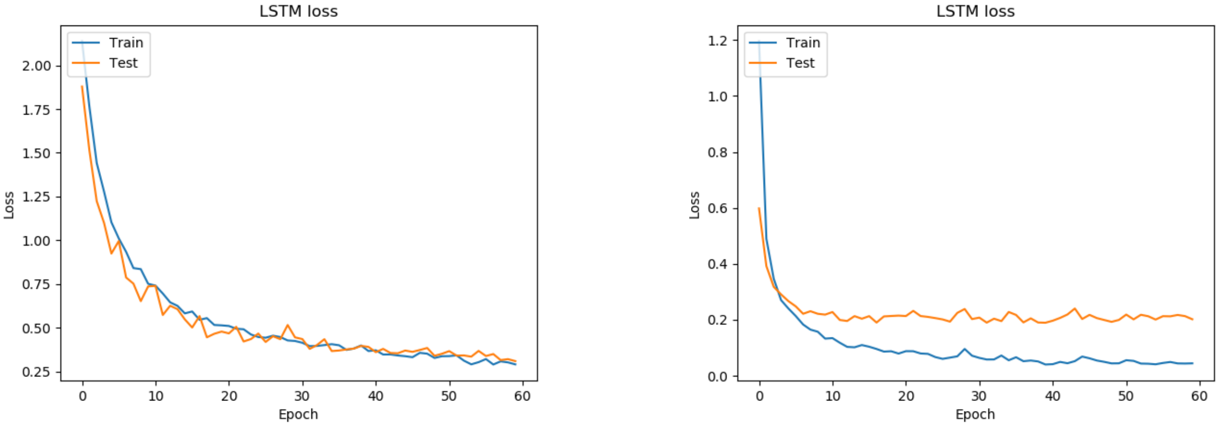
Figure 13: Training and validation loss of RNN model.



(a) LSTM's accuracy using SGD.

(b) LSTM's accuracy using Adam.

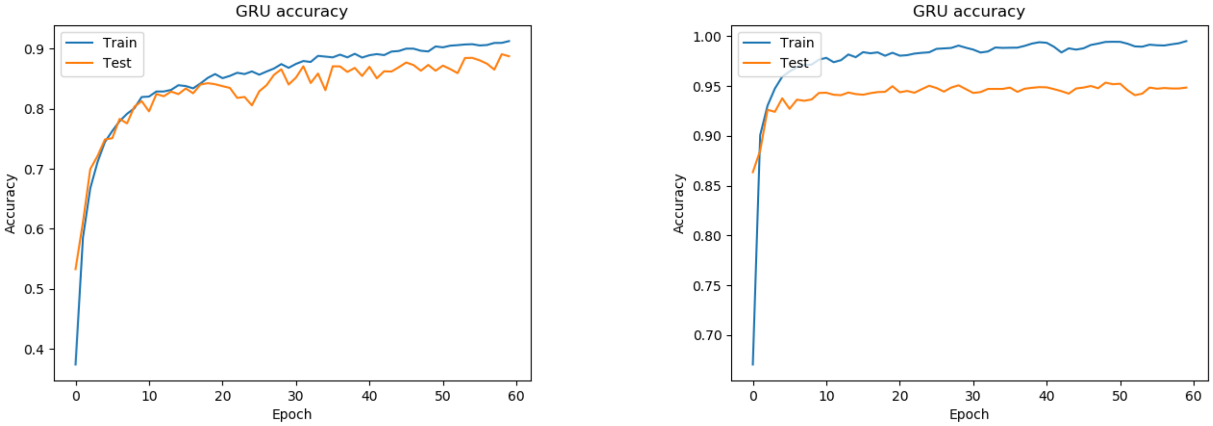
Figure 14: Accuracy comparison between LSTM using SGD and Adam.



(a) LSTM's loss using SGD.

(b) LSTM's loss using Adam.

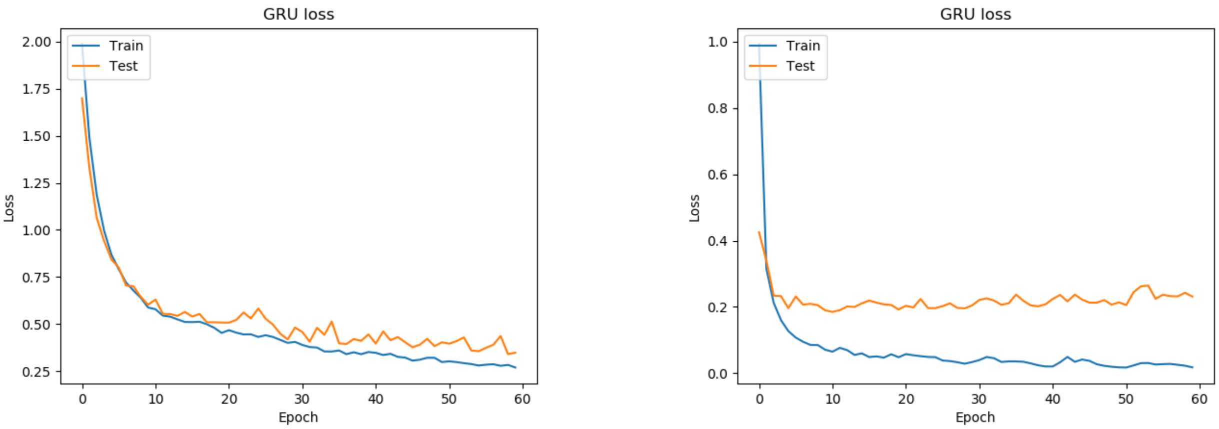
Figure 15: Loss comparison between LSTM using SGD and Adam.



(a) GRU's accuracy using SGD.

(b) GRU's accuracy using Adam.

Figure 16: Accuracy comparison between GRU using SGD and Adam.



(a) GRU's loss using SGD.

(b) GRU's loss using Adam.

Figure 17: Loss comparison between GRU using SGD and Adam.

Training on Luxembourghish dataset.

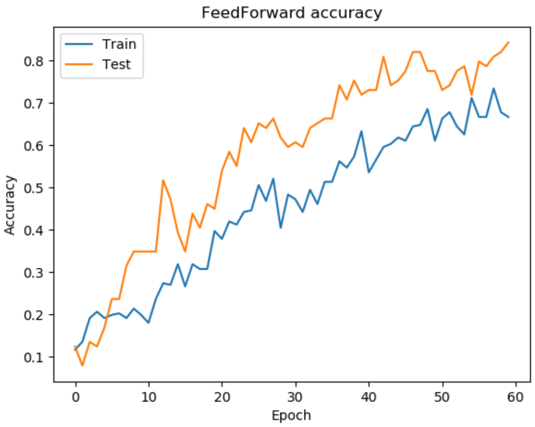


Figure 18: Training and validation accuracy of feedforward model trained on Luxembourgish dataset.

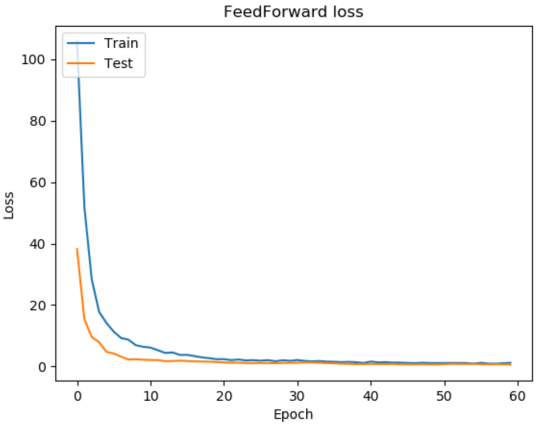


Figure 19: Training and validation loss of feedforward model trained on Luxembourgish dataset.

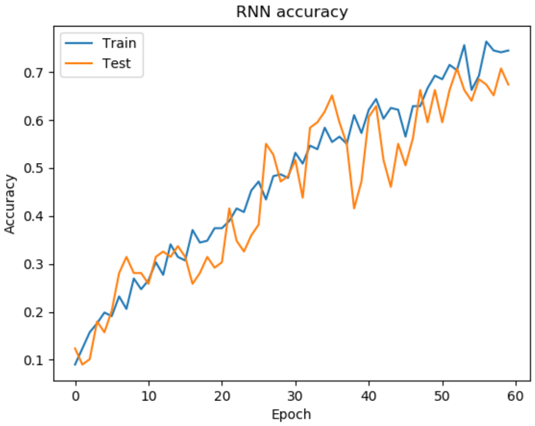


Figure 20: Training and validation accuracy of RNN model trained on Luxembourgish dataset.

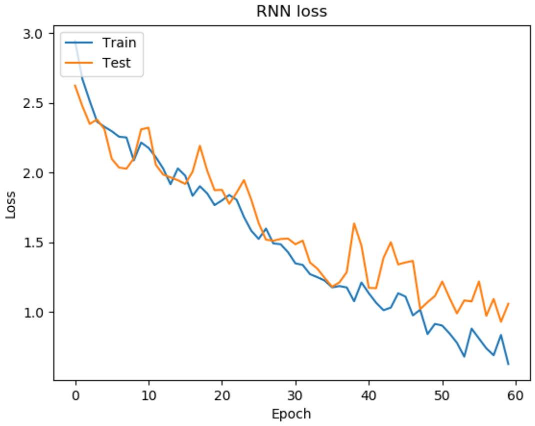


Figure 21: Training and validation loss of RNN model trained on Luxembourgish dataset.

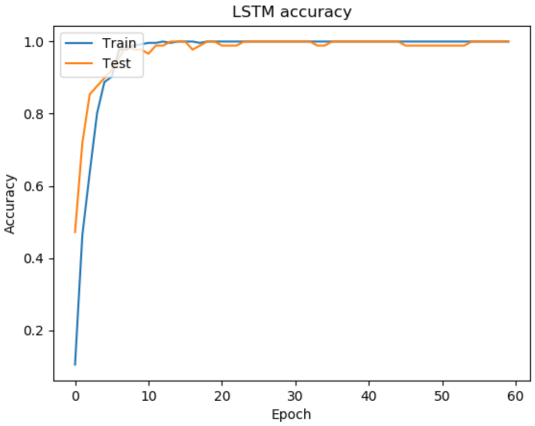


Figure 22: Training and validation accuracy of LSTM model trained on Luxembourgish dataset.

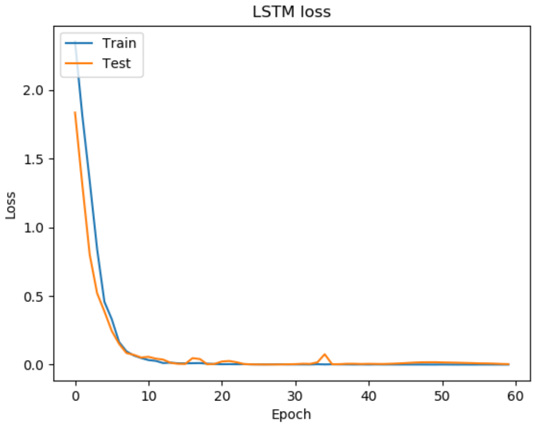


Figure 23: Training and validation loss of LSTM model trained on Luxembourgish dadataset.

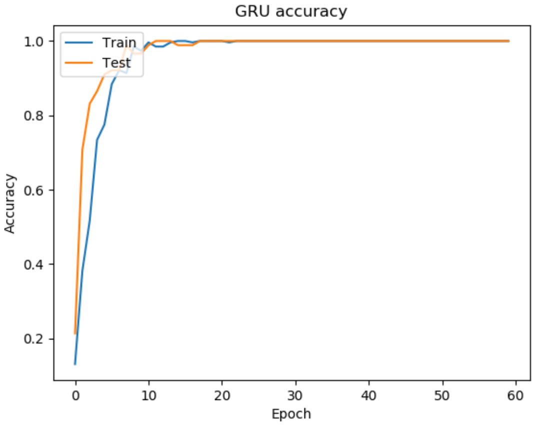


Figure 24: Training and validation accuracy of GRU model trained on Luxembourgish dadataset.

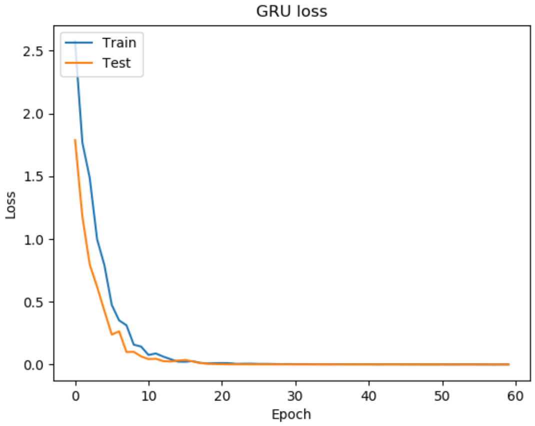


Figure 25: Training and validation loss of GRU model trained on Luxembourgish dadataset.

```
import librosa
import numpy as np
import pandas as pd
import os
import csv

numbers = ’0 1 2 3 4 5 6 7 8 9’.split()

header = ’’
for i in range(1, 641):
    header += f’mfcc{i} ’
header += ’ label’
header = header.split()

file = open(’largetrainingset.csv’, ’w’, newline=’’)
with file:
    writer = csv.writer(file)
    writer.writerow(header)

for number in numbers:
    for filename in os.listdir(f’./recordings/{number}’):
        nfile = f’./recordings/{number}/{filename}’

        y, sr = librosa.load(nfile, sr=None, mono=True, duration=1)

        if y.size < 16000:
            rest = 16000 - y.size
            left = rest // 2
            right = rest - left

            y = np.pad(y, (left, right), ’reflect’)

        mfccs = librosa.feature.mfcc(y=y, sr=sr)

        row = mfccs.T.flatten()
        row = np.append(row,number)

        file = open(’largetrainingset.csv’, ’a’, newline=’’)
        with file:
            writer = csv.writer(file)
            writer.writerow(row)
```

Figure 26: Code snippet to extract MFCCs from the dataset.