KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Building an end-to-end speech recognizer

Moritz Wolter

Thesis submitted for the degree of Master of Science in Mathematical Engineering

Thesis supervisors:

Prof. dr. ir. Johan Suykens
Prof. dr. ir. Hugo Van hamme

Assessors:

Prof. dr. ir. Raf Vandenbrid
Prof. dr. ir. Patrik Wambacq

Mentor:

Ir. Vincent Renkens

Academic year 2016 – 2017

Preface

This thesis is about attention mechanisms in speech recognition. Attention allows artificial neural nets to focus on audio, video or text data. The concept is therefore not only important in speech processing. It also appears in amazing feats of engineering such a neural Turing machines [17], automatic translators [2] or self programming computers [24]. I hope you will enjoy reading about this exciting topic.

I would like to thank my supervisor Vincent Renkens, for introducing me to the subject and the constant stream of good ideas and code he contributed to this project. I would like to extend my gratitude to my supervisor Prof. Van hamme for his feedback, debugging ideas, and for believing into this project even when nothing was working. Furthermore would like to thank everyone at the speech group in Leuven for their feedback during numerous lunch meetings.

My sincere gratitude also goes to my parents for supporting me all the way trough my studies.

Moritz Wolter

Contents

Abstract

Speech recognition is concerned with transcribing what is said in a recoding of spoken language. In machine learning terms this process is called sequence labeling. A recoding consists of a chain of frames, this chain can be split up into several sequences, these make up words or phonemes, which must be labeled. The sequence of labels forms the transcription.
The meaning of speech depends on context, therefore a good system needs to take it into account. Classical feed-forward networks fail to do that, which is why this system will mainly consist of recurrently connected Long Short Term Memory (LSTM) blocks. Inspired by the recurrent connects of neurons in the human brain, LSTM-RNNs have the ability to store information over long time periods, an important requirement in take context into account.
In order to train machine learning systems, speech and transcription text pairs are used. The text contains the exact information of what is said in the recording, but where in the recoding which word or sound is said is unknown. In other words text to speech alignment is missing. They key problem in this thesis is data alignment. Potential solutions such as connectionist temporal classification as well as attention based transducers will be discussed, with the latter being the main concept of this thesis.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

ASR	Automatic speech recognition
MLP	Multilayer perceptron
CNN	Convolutional neural network
RNN	Recurrent neural network
LSTM	Long short term memory
BLSTM	Bidirectional long short term memory
PLSTM	Pyramidal long short term memory
LAS	Listen attend and spell

Symbols

$\mathbf{m}$	Mean vector
σ	Standard deviation vector
$\mathbf{x}$	System input vector
$\hat{\mathbf{x}}$	Standardized input vector
$\hat{\mathbf{y}}$	Label probability distribution vector
$\mathbf{y}$	Label output vector
E	Cost function
$\mathbf{w}$	Model weight vector
$\triangle$	Gradient
$\mathbf{c}$	Recurrent cell state and las context
$\mathbf{h}$	Hidden value vector
α	Attention vector

Chapter 1

Problem statement

Automatic speech recognition is concerned with finding ways to enable computers to recognize spoken language and transcribe it to text. Speech recognition asks the question:
What is being said?

In order to answer this question one must determine, which parts of a recording contain relevant information. These parts should then be decoded and the rest ignored. In order to answer the first question one must ask a second:
Which parts are interesting in a recording?

If interesting parts are found in the recording the system should label them correctly. During the labeling process a sequence of inputs is assigned a sequence of labels. Following this train of thought a sequence to sequence labeling problem must be solved. Speech data consists of frames. Transcription means grouping the interesting frames of the input sequence and assigning labels to these groups. Once input sequences groups are found and matched with a label sequence, the two are considered to be aligned. In order to do the alignment one must know:
How can sequence to sequence alignment be established?

This thesis relies on machine learning methods in its attempt to determine what is being said. Machine learning models typically consists of many unknown weights, which are initially chosen at random. The good values for each weight are determined using a form of gradient descent. The process of using gradient descent to determine good model parameters is often referred to as training. Unfortunately gradient descent does not always lead to an acceptable solution. Only if the training algorithm is run with carefully chosen hyper-parameters on a model complex enough form an internal representation of the patters it is trained to extract, the optimization process will terminate at a good optimum. Using machine learning methods leads to another important question:
What model architecture is capable of handling the complex patterns found in speech data? Which hyper-parameters should be chosen to train such a model?

A well known problem of machine learning methods is over-fitting, one must therefore to ask:
How can a model that generalizes well be trained?

Working with raw speech data should be possible in principle, in the speech literature researchers often use frequency domain representations of the original speech data. These representations are also called features, which leads to another question:
What kind of feature representation of the input signal should be used if any?

At this point there are a lot of open questions and finding answers might not always be easy, it would therefore be interesting to know:
Which methods will allow the solution of any of the questions above?

Last but not least, this thesis should be useful to others. It's code contributions should benefit future researchers, which leads to one last question:
How should software be developed in order to produce useful and maintainable results?

This thesis is an attempt answer the questions above, trough literature study, coding and experimentation.

Chapter 2

Literature Study

In this chapter contains a review of some of the most important techniques necessary to build an end-to-end speech recognition system. After looking at possible input feature representations, the most important machine learning tools will be explored. Finally sequence-to-sequence labeling methods such as CTC and the listen attend and spell architecture are covered.

2.1 Preprocessing and feature extraction

Filter-Bank features

Filter-banks are collections of filters. These filters can be spread out over audible frequencies[1]. Filter-bank output is commonly used as input for speech analysis [20][7]. The number of filter-banks depends on the required resolution, 32 is a common choice [21]. The energy within the part of the signal spectrum described by all individual filters is measured. Figure 2.1 shows the resulting energy measurements using 23 filters, for a sentence recording contained in the *TIMIT* data set. The general argument for filter banks in speech recognition is that the cochlea, in the human ear, resembles a filter bank [20, page 30]. Humans do not perceive frequency linearly. Experimental evidence suggests, that our perception of is scaled according to the Mel-Scale [20, page 34]:

$$B(f) = 1125ln(1 + f/700) \tag{2.1}$$

A normalized plot of this function is shown in figure 2.2 on the left. According to the Mel-scale, humans are able to distinguish more lower frequencies than higher

[1] Approximately 16 to 16000 Hz.

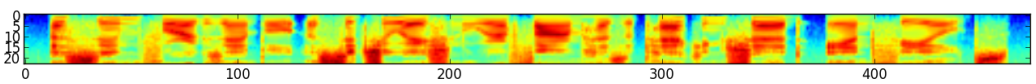


Figure 2.1: Frequency Bank input computed from a sentence contained in the *TIMIT* dataset. Time is shown on x and Frequency on the y-Axis.

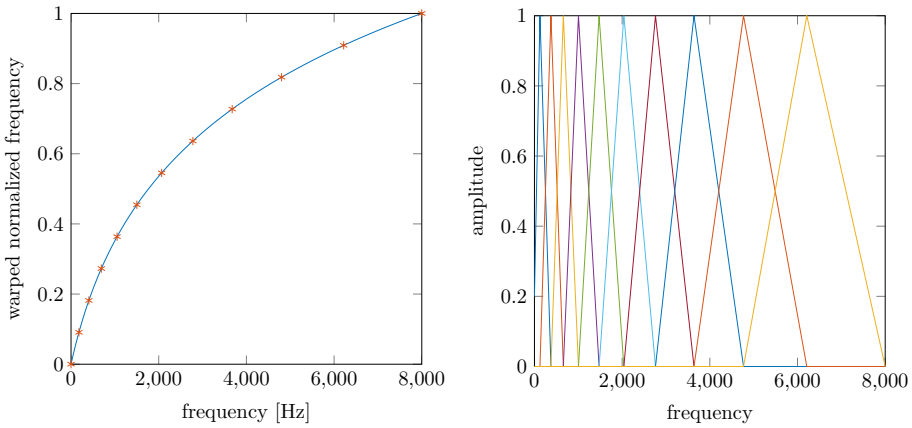


Figure 2.2: The Mel-scale (blue) with Mel-Frequency Cepstrum Coefficients (red) on the left. Filterbank with Mel-spaced filters (right).

frequencies. In the plot the first four thousand Herz occupy roughly eighty percent of the scale. The band from four thousand to eight thousand Herz is left with only about twenty percent of the scale, even tough half of the considered frequencies are in this band. Mel spaced filter-banks are an attempt to include the human perception in speech recognition. The filter functions are defined by [20, page 317]:

$$H_m = 0 \qquad \text{if } k < f[m-1] \tag{2.2}$$

$$H_m = \frac{k - f[m-1]}{f[m] - f[m-1]} \qquad \text{if } f[m-1] \leq k \leq f[m] \tag{2.3}$$

$$H_m = \frac{f[m+1] - k}{f[m+1] - f[m]} \qquad \text{if } f[m] \leq k \leq f[m+1] \tag{2.4}$$

$$H_m = 0 \qquad \text{if } k > f[m+1] \tag{2.5}$$

In the equations above H_m denotes the magnitude of filter m with a total of M filters. The frequency is denoted by k, the vector f contains $M + 2$ linearly spaced filter border values. These are the red stars on the left of figure 2.2. The right plot shows the triangular filter banks. These banks are spaced according to the same values. Roughly speaking using mel-filter banks means using a high filter resolution where human hearing is good and a low resolution where it is bad.
In addition to perceptually motivated instantaneous features, derivative information can be appended to each feature vector. These so called delta features add information regarding the speed of change taking into account $2 * N$ feature vectors. N denotes the number of frames back and forward in time. The derivative information can be computed using [30]. The mel feature vectors are denoted by $\mathbf{c}$:

$$\mathbf{d}_t = \frac{\sum_{n=1}^{N} n(\mathbf{c}_{t+n} - \mathbf{c}_{t-n})}{2 \sum_{n=1}^{N} n^2} \tag{2.6}$$

The formula above is a central difference, because the value at time t is not taken into account, in contrast to forward or backward differences, where the value at t is part of the difference. A common choice for N is two. In this case the formula above simplifies to:

$$\mathbf{d}_t = \frac{2\mathbf{c}_{t+2} + \mathbf{c}_{t+1} - \mathbf{c}_{t-1} - 2\mathbf{c}_{t-2}}{10} \tag{2.7}$$

In this case two future frames are used to compute the derivative at time t, which implies, that at computation time a full recording is available. Should that not be the case a backward difference scheme should be considered instead. To further augment the features second derivative or acceleration information can be added as well. These can be computed by simply applying the central difference to the deltas one more time. It is important to note that augmenting the features with derivative information significantly increases their dimension. If 40 mel-filters are used then the first and second derivatives will contain 40 entries each, therefore increasing the input dimension from 40 to 120.

Before using the features it is best practice to standardize the components of the input vectors. It is desirable to use features with an overall mean of zero and a standard deviation of 1 over the entire training set [12, page 30]. The feature normalization process starts by computing the mean,

$$\mathbf{m}_i = \frac{1}{\|S\|} \sum_{\mathbf{x} \in S} \mathbf{x}_i \tag{2.8}$$

and standard deviation

$$\sigma_i = \sqrt{\frac{1}{|S| - 1} \sum_{x \in S} (\mathbf{m}_i - \mathbf{x}_i)^2} \tag{2.9}$$

Of a data set S consisting of input $\mathbf{x}$ and target $\mathbf{t}$ pairs. Standardized input vectors can then be computed using:

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mathbf{m}_i}{\sigma_i} \text{ for all } \mathbf{x}_i \in S. \tag{2.10}$$

Input standardization is beneficial to network performance, because it puts the input values in ranges, which standard activation functions such as sigmoids or hyperbolic tangents can handle better [12, page 30].

Mel-Frequency banks are considered high level feature inputs. When the recognition system is found to work with these, features on a lower level or even raw data could be used as input. The idea behind doing less preprocessing, is that the network might be able to come up with something better on its own.

2.2 Neural Networks

Artificial neural networks where originally designed to model biological neurons. Even tough it is now known that artificial neural networks have little in common

with their counterparts in biology, the models are still popular due to their useful pattern recognition properties [12, page 13]. All neural networks consist of elementary functions with weighted and biased inputs, the operations necessary to evaluate them form a computational graph. In a nutshell the idea is to choose the weights and biases such that the network output corresponds to patterns seen in the input data. The process of finding good weights is called training. Training in turn is done using gradient descent.

2.2.1 Gradient descent

The optimization process of neural networks works with a training data set $\{\{\mathbf{x}_1, \mathbf{t}_1\}, \ldots, \{\mathbf{x}_p, \mathbf{t}_p\}\}$ [26, page 156]. The elements of this set are the input- and output-patters $\mathbf{x}$ and $\mathbf{t}$ respectively. For each input vector the network output $\mathbf{o}$ is computed. Ideally the network output should be the same as the desired target $\mathbf{t}$ one for all data pairs. The difference between target and current outputs could be measured by the cost function [26, page 156]:

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\mathbf{o}_i - \mathbf{t}_i\|^2. \tag{2.11}$$

However generally the more complex cross-entropy is used as a cost function, because it leads to faster convergence and better results in practice [11]. For classification tasks, cross entropy is defined as [6, page 245]:

$$E = -\sum_{i=1}^{p} \mathbf{t}_i \ln(\mathbf{o}_i) + (1 - \mathbf{t}_i) \ln(1 - \mathbf{o}_i) \tag{2.12}$$

In order to understand why the cross entropy is a measure of output error, the two scalar cases $t = 0$ and $t = 1$ are considered. For $t = 0$ the cross entropy simplifies to $-\ln(1 - o)$. Looking at figure 2.3, this expression turns out to be a good cost function, because for values close to the desired output $o = 0$, $-\ln(1 - o)$ will be close to zero. If the output moves away from the desired value zero, the cost grows asymptotically. If $t = 1$ is considered the cost function simplifies to $-\ln(o)$ which will be zero for the desired output and once more displays asymptotic growth for undesired values far away from one. If the target probability is somewhere between these two extreme cases the cost will be the sum of the two cases considered earlier. Cross entropy is statistically motivated, the output vectors $\mathbf{o}$ describe probability distributions over the possible labels, their elements must therefore be $\in (0, 1)$ and sum op to one.

During the training process a local minimum of the error function E is sought. At this minimum the difference between the network output $\mathbf{o}$ and the target values $\mathbf{t}$ is small. This value might not be the smallest possible value, termination can happen at the global or a local minimum. After the training process has completed the network is expected to identify similarities to data seen during the training process and produce a similar output. In order to reach a local minimum, the gradient of the error function is needed. The key idea of gradient descent is to follow

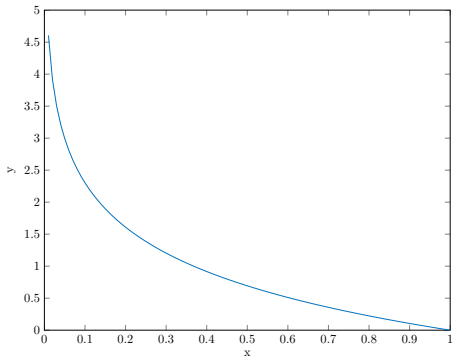


Figure 2.3: Plot of $y = -\ln(x)$ for $x \in (0, 1)$.

the negative gradient until a local minimum is reached. Neural networks can be considered as large composite functions, which are made up of elementary operations. The evaluation of the network can be written as a graph. Computations are done at each node and information travels trough the network along directed edges from node to node. In order to create a computational graph for the training process each of the output units of the network under consideration are connected to a new node which computes $\frac{1}{2}(o_{ij} - t_{ij})^2$[26, page 157] or the cross-entropy term of one data target pair. These new nodes in turn are connected to one more node, which sums up all error values and produces E_i. The process described above must be repeated for all training data pairs. One final node is added, which sums up all values E_i. Its output gives the value for the error function E which is now in the form of a large graph.
Reverse mode algorithmic differentiation or back-propagation is an algorithm to compute the gradient of a graph consisting of basic elementary operations. As an example its operation is now illustrated using the function [9, page 69]:

$$f(w_1, w_2, w_3) = \sin(w_1 w_2) + \exp(w_1 w_2 w_3) \tag{2.13}$$

This function written in terms of five elementary operations as [9, page 70]:

$$w_4 = w_1 w_2 \tag{2.14}$$
$$w_5 = sin(w_4) \tag{2.15}$$
$$w_6 = w_4 w_3 \tag{2.16}$$
$$w_7 = exp(w_6) \tag{2.17}$$
$$w_8 = w_5 + w_7 \tag{2.18}$$
$$o = w_8 \tag{2.19}$$

Computing the gradient means computing the partial derivatives of f with respect

to all inputs. In this case this means finding:

$$\frac{\partial f(w_1, w_2, w_3)}{\partial w_1} = w_2(\cos(w_1 w_2) + w_3 \exp(w_1 w_2 w_3)) \tag{2.20}$$

$$\frac{\partial f(w_1, w_2, w_3)}{\partial w_2} = w_1(\cos(w_1 w_2) + w_3 \exp(w_1 w_2 w_3)) \tag{2.21}$$

$$\frac{\partial f(w_1, w_2, w_3)}{\partial w_3} = w_1 w_2 \exp(w_1 w_2 w_3) \tag{2.22}$$

Above the derivatives have been found by hand using the chain rule. Now these will be computed using back-propagation. Figure 2.4 shows a graphical representation of equation 2.13. The partial derivatives needed for the backwards sweep can be found on the edges.

The gradient is computed using a forward and backward sweep. During the forward sweep the inputs are fed into the network and the functions at each node are evaluated layer by layer, until the output at the last node is known. In figure 2.4 this means computing w_4 to w_8.

After the forward sweep the gradient is found by going back trough the network from the output to each input node. Using a seed value of 1 at the output node the lower unit values are computed by multiplying the associated partial derivative found on each edge. If a node has more then one incoming value, their sum is computed. The process is illustrated in figure 2.5. At the roots of the tree the partial derivatives of the output with respect to each input can be found. Together these root values make up the gradient. To be able to perform the first forward sweep the network weights are initialized at random. The training data pairs are known and can be added as constants to the graph. If the weights of the network are stored in a weight vector $\mathbf{w}$ the value of the gradient after the nth update may be written as [12, page 27]:

$$\triangle \mathbf{w}(n) = -\alpha \frac{\partial f}{\partial \mathbf{w}(n)}. \tag{2.23}$$

Where $\alpha \in [0, 1]$ denotes the learning rate. Unfortunately simple gradient descent tends to get stuck in local optima. In order to increase the chance of gradient descent to escape from such a local minimum a momentum term can be added to the formulation [6, page 267][12, page 27]:

$$\triangle \mathbf{w}(n) = m \triangle \mathbf{w}(n-1) - \alpha \frac{\partial f}{\partial \mathbf{w}(n)}. \tag{2.24}$$

Above $m \in [0, 1]$ denotes the momentum. In order to understand the effect of the momentum term consider a weight space region of very low curvature, where it can be assumed, that the gradient stays constant. Using the momentum formula above yields [6, page 267]:

$$\triangle \mathbf{w}(n) = -\alpha \frac{\partial f}{\partial \mathbf{w}}(1 + m + m^2 + \dots) \tag{2.25}$$

$$= -\frac{\alpha}{1-m} \frac{\partial f}{\partial \mathbf{w}}. \tag{2.26}$$

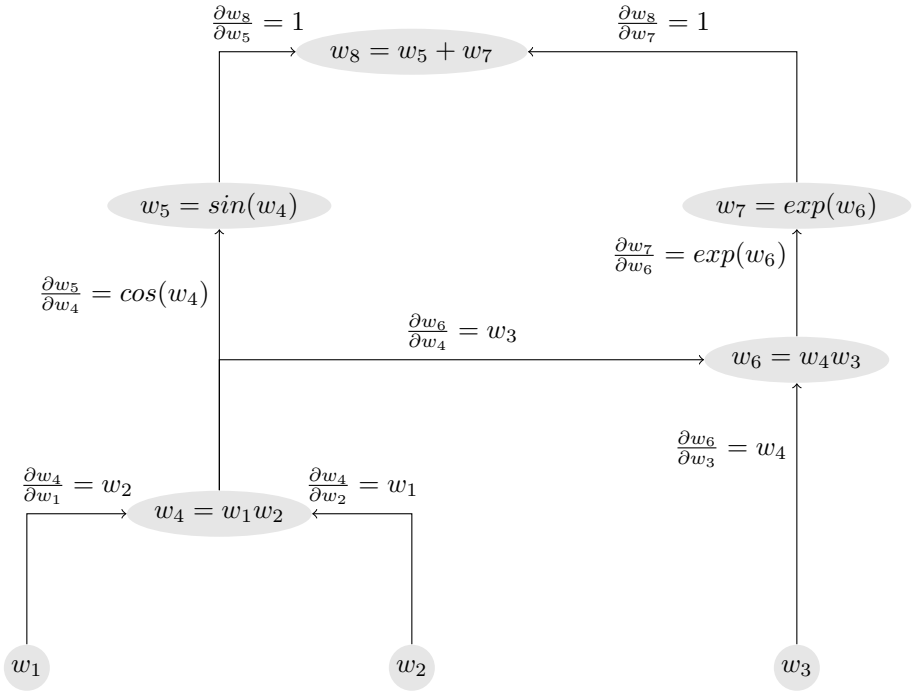


Figure 2.4: Example funciton network with partial derivatives.

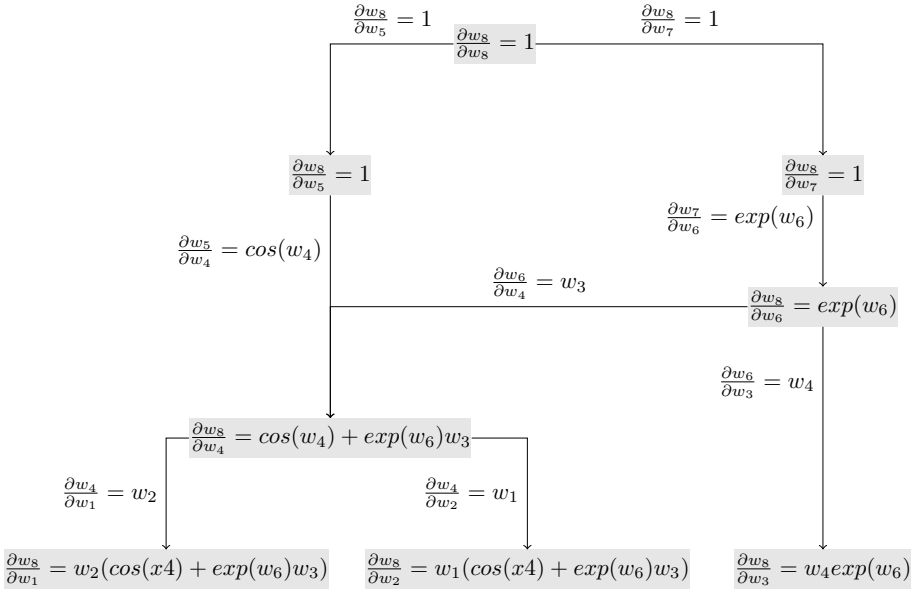


Figure 2.5: Reverse sweep.

Using the fact that the sum of the geometric series is known and $|m| < 1$ must hold. The momentum term therefore changes the effective learning rate from α to $\frac{\alpha}{1-m}$ in regions of low curvature. Elsewhere the gradient is oscillatory, which leads to cancellation of successive contribution of the momentum term [6, page 267].

2.2.2 Regularization

The optimization process works on the training set, but later the trained network should be able to recognize patterns seen in the training data set in other inputs and produce similar outputs. In order to achieve this goal it is important not to memorize training data samples. Another problem with complex networks is that they are able to track noise in the training data. Regularization techniques aim to counteract sample memorization and noise tracking in order to ensure that progress on the training data carries over to new data sets or in other words regularization aims to improve generalization.

Early stopping

The idea of early stopping is to observe the training and validation loss during training. As long as both are falling simultaneously, we assume that the network will also generalize well to other data. This is a reasonable assumption, because the validation data set plays no part in computing the weight updates. When the training set loss continues fall, but the validation loss rises, the network is overfitting [13, page 31]. The training process should be stopped early in theses cases. A test set should be used to guard against parameter choices that work well only on the training and validation set. Ideally the performance on the test set should only be evaluated, once the training process has finished.

Care must be taken, when dividing the entire data set into training, validation and test subsets. All three sets must be good representations of the overall data set. Differences for example regarding the noise level should be avoided. Furthermore the sizes of all sets must be chosen carefully. Using too few samples for validation and testing, makes it very hard to guard against over-fitting. If the training set becomes to small, overall performance might suffer. In practice a good trade off must be found experimentally.

When data collection is expensive cross validation should be used. Cross validation divides the merged training and validation data sets into k subsets. During each iteration one of the k subsets is used for validation and the remaining $k-1$ sets are used for training. Cross-validation has the advantage that each subset is used for training and validation, thus making more efficient use of the total data set. The test set is very important when working with cross validation, because the validation data is part of the training process. It should be sufficiently large to detect over-fitting.

Input Noise

Training with input noise means that fixed variance, zero mean Gaussian noise is added to the inputs during training [13, page 32]. The targets remain unchanged.

The idea is to reuse the training data set more efficiently. When the artifical noise is similar to the noise already present in the data set, reusing the noisy samples is better than reusing the original samples. Input noise decreases the reliability of the inputs during training, is covers small variations caused by noise in the data, which the network could remember without it during training, if the noise was not present. Furthermore the artificial noise will help the network to deal with similar noise in the future, making the trained system more robust. A problem with input noise is that it is difficult to determine how large the variance should be, one option is to manually tune the parameter using the validation performance as a metric.

Weight noise

When using weight noise zero mean fixed variance Gaussian noise is added to the weights during training. The idea is to reduce the precision, with which the network weights can be described [13, page 32]. The reduced precision makes it impossible for the network to model small variations in the training samples, which are likely to be noise. Instead it will only be able to model larger changes and generalize better. When using input noise the following algorithm should be used [13, page 33]:

```
while stopping condition is False:
    Shuffle training data pairs
    for each training sample pair:
        Add Gaussian noise to weights
        compute the gradient
        restore original weights
        update the original weights
```

However weight noise should be used with care, because it can lead to very slow convergence.

Regularized loss minimization

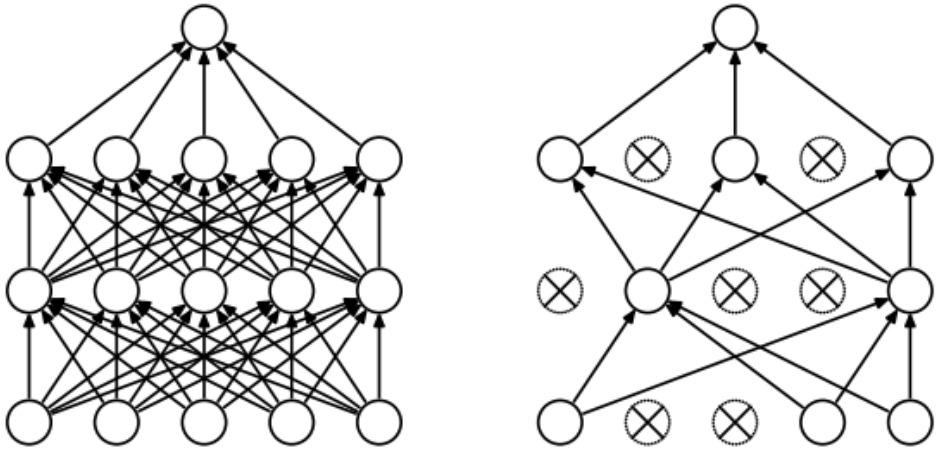
Regularized loss minimization includes a model complexity penalty term in the cost function. Instead of minimizing the loss E an augmented loss [6, p. 338][3, p. 171]:

$$\hat{E} = E + v\Omega \tag{2.27}$$

is used. Ω stands for a penalty term, which should grow, when the network is likely to over-fit the training data. $v \in (0, 1)$ denotes the weight parameter, which determines how much emphasis the optimization algorithm should place on regularization. A straightforward choice for Ω is the l_2 norm of the model weights:

$$\Omega_{l2} = \|\mathbf{w}\|^2 = (\sqrt{\sum w^2})^2 \tag{2.28}$$

The weight norm serves as a measure for the model complexity. The idea of choosing the simplest possible model which can explain the data, often referred to as Occam's razor is expressed in the modified cost function $\hat{E}$, the result of the optimization process will be a compromise between fit to the training data and model complexity.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.6: "Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped." [27], image and caption taken from the same source.

Dropout

Like input noise and weight noise dropout is part of the group of regularization techniques, that artificially corrupt training data to improve prediction performance [29]. When applying dropout between two layers, some units and their connections are zeroed out at random during training. The process is illustrated in figure 2.6. One dropout interpretation relates dropout networks to committee networks. Network committees consist of several independently trained networks, each of which makes predictions on its own. The committee prediction is then produced by using the mean output of the committee members or by feeding the committee outputs into another neural net, which decides on the final output. When training large deep networks however committee networks become infeasible due to their large memory requirements. Dropout networks are related to committee networks, because when some units are dropped the remaining units will have to learn to work with the remaining units. Just like committee networks, dropout networks contain multiple mechanism to recognize the same pattern, and thus gains some of the prediction robustness that committee networks are know to have. Another motivation for dropout comes from the theory of sexual selection [27]. The theory assumes that the superiority of sexual reproduction is caused by the fact that removes strong interconnections between genes. After all a specific gene might not be there a generation from now. To function over many generations, genes must therefore develop redundant ways that work with different genes. Following this interpretation dropout forces network units to do the same thing.

2.2.3 Deep Neural Networks

Deep learning deals with the design of computational nets, consisting of multiple layers, which learn representations of data. Deep networks are often used when abstract patterns must be found. Simple networks are often unable to cope with the problem complexity, and fail to recognize the underlying structure. Human speech features such complex patterns, therefore deep networks must be considered. Increasing the network depth leads to many additional weights, for which a suitable value must be found. In addition to the added memory load caused by the extra weights, the optimization process must look trough large data sets until a suitable optimum is reached.

Stochastic gradient descent

When training deep networks on very large data sets, working with the full data set to compute the current gradient becomes very inefficient. It is common practice to use stochastic gradient descent instead of the classic algorithm. The idea is to select small subsets of the full data set. These so called mini-batches are then used to compute the outputs and errors, as well as the gradients for these smaller examples. Instead of working with the gradient of the union of these mini-batches the average gradient is computed and used to update the network weights. This approach dramatically reduces the memory requirements of the training process, which makes it possible to work with more sophisticated network designs.

2.2.4 Recurrent Neural Networks

When processing speech its is important to take context into account. When spelling the letters which make up a word, it is important to know what the previous letter was, in order to make the right decision. Feed-forward neural nets do not possess memory. These networks make decisions, starting from zero every time. In oder to fix this a cell state variable can be introduced. A simple recurrent cell where the current state is set to the previous output is shown in figure 2.7. Another way to depict the same architecture is shown in Figure 2.8, here the cell state is not labeled explicitly, the single cell on the left is just a simplified version of the figure above. If the network is unrolled in time the flow of the state becomes apparent, which is done in figure 2.8 on the right. The unrolled form shows a direct dependency of the output at time t on the previous output at $t-1$, which in turn depends on the previous outputs. This causes y_t and y_{t-1} to change together. In other words: The introduction of recurrent connections leads to correlation of the two outputs.

The exploding and vanishing gradient problem

Even tough past information is available in theory, RNNs have only limited access to contextual information in practice [12, page 1]. Due to problems with gradient descent on correlated data, the back-propagated derivative can sometimes become weaker and weaker until it ultimately vanishes [18]. Another problem is that sometimes

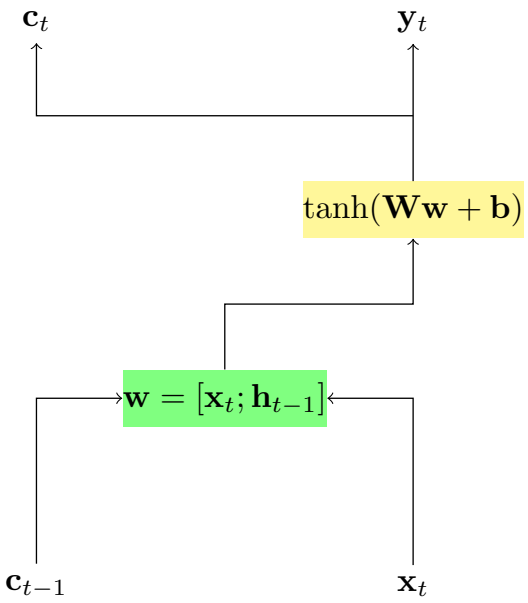


Figure 2.7: Visualization of a single recurrent cell.

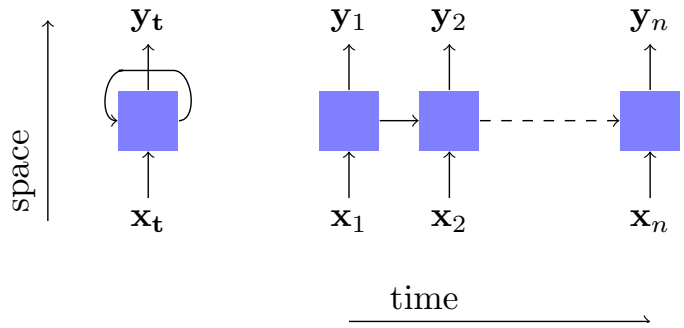


Figure 2.8: Rolled (left) and unrolled (right) recurrent neural net with two units.

classical recurrent neural nets produce a gradient that blows up [25]. The exploding gradients can be fixed by clipping, but vanishing gradients require more sophisticated treatment [5].

Long short-term memory

Initially researchers tried to solve the vanishing gradient problem by making changes to the training algorithm using simulated annealing, time delays or compression [12, page 32]. However a good solution to the problem turned out to be changing the RNN cell architecture. Long short-term memory (LSTM) cells as proposed in [19] are more complex network units. LSTMS are differentiable versions of memory chips in a digital computer. Memory chips generally have read, write, and erase ports which can be set in order to allow the chips state to be read, modified or emptied. In the LSTM case the input, output, and forget gates serve the same function [12, page

33]. Throughout the literature these gates are generally denoted as $\mathbf{i}, \mathbf{f}$ and $\mathbf{o}$. The content of the memory or state is written as $\mathbf{c}$, the output as $\mathbf{h}$, while a t subscript denotes the time step. These memory cells use the differentiable equation system [14, page 5][2]:

$$\mathbf{i_t} = \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h_{t-1}} + \mathbf{W}_{ic}\mathbf{c_{t-1}} + \mathbf{b}_i) \tag{2.29}$$
$$\mathbf{f_t} = \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h_{t-1}} + \mathbf{W}_{fc}\mathbf{c_{t-1}} + \mathbf{b}_f) \tag{2.30}$$
$$\mathbf{c_t} = \mathbf{f_t}\mathbf{c_{t-1}} + \mathbf{i_t}\tanh(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{ch}\mathbf{h_{t-1}} + \mathbf{b}_c) \tag{2.31}$$
$$\mathbf{o_t} = \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h_{t-1}} + \mathbf{W}_{oc}\mathbf{c_t} + \mathbf{b}_o) \tag{2.32}$$
$$\mathbf{h_t} = \mathbf{o}_t\tanh(\mathbf{c}_t) \tag{2.33}$$
$$\tag{2.34}$$

From the definition of the matrix product follows that

$$\mathbf{A}\mathbf{x}_1 + \mathbf{B}\mathbf{x}_2 = \begin{bmatrix}\mathbf{A} & \mathbf{B}\end{bmatrix} \cdot \begin{bmatrix}\mathbf{x}_1 \\ \mathbf{x}_2\end{bmatrix}. \tag{2.35}$$

Which this relation in mind the equations above can be rewritten, by creating column wise concatenated weight matrices for every neuron gate W_i, W_f, W_o, as well as for the state W_c. These matrices can then be multiplied by a row wise concatenated vector $[\mathbf{x}_t\ \mathbf{h_{t-1}}\ \mathbf{c}]^T$, which leads to the slightly simplified system of equations below:

$$\mathbf{i_t} = \sigma(\mathbf{W}_i[\mathbf{x}_t\ \mathbf{h_{t-1}}\ \mathbf{c_{t-1}}]^T + \mathbf{b}_i) \tag{2.36}$$
$$\mathbf{f_t} = \sigma(\mathbf{W}_f[\mathbf{x}_t\ \mathbf{h_{t-1}}\ \mathbf{c_{t-1}}]^T + \mathbf{b}_f) \tag{2.37}$$
$$\mathbf{c_t} = \mathbf{f}_t\mathbf{c_{t-1}} + \mathbf{i}_t\tanh(\mathbf{W}_c[\mathbf{x}_t\ \mathbf{h_{t-1}}]^T + \mathbf{b}_c) \tag{2.38}$$
$$\mathbf{o_t} = \sigma(\mathbf{W}_o[\mathbf{x}_t\ \mathbf{h_{t-1}}\ \mathbf{c_t}]^T + \mathbf{b}_o) \tag{2.39}$$
$$\mathbf{h_t} = \mathbf{o_t}\tanh(\mathbf{c_t}) \tag{2.40}$$

This system of equations is visualized in figure 2.9. Just like diagram 2.7 this depiction is read from bottom to top. The most important part is the line from $\mathbf{c}_{t-1}$ to $\mathbf{c}_t$ [8]. It records operations on the cell state $\mathbf{c_t}$. The cell state contains information from the past which helps the block make decisions regarding the current output $\mathbf{h}_t$. The sigmoid functions $\sigma(\cdot)$ are applied element wise on the input vectors and produce outputs between zero and one. In the case of the forget gate output $\mathbf{f}_t$ these values $\in (0, 1)$ well serve as a measure of how much of the past state the cell would like to remember. One means keep this variable and zero throw it away [8]. The following task is to determine what should be added to the memory. This information can be found in the input gate result $\mathbf{i}_t$. $\mathbf{i}_t$ is multiplied element wise with the candidate values $\bar{\mathbf{c}}_t$. These are computed by a hyperbolic tangent neuron. The $\tanh(\cdot)$ function makes sure all vector elements are between -1 and 1. The neuron computing the candidate state values $\bar{\mathbf{c}}_t$ looks at input data and the past outputs.

[2] Various versions of LSTM cells exist. This one is commonly referred to as the "peephole" variant.

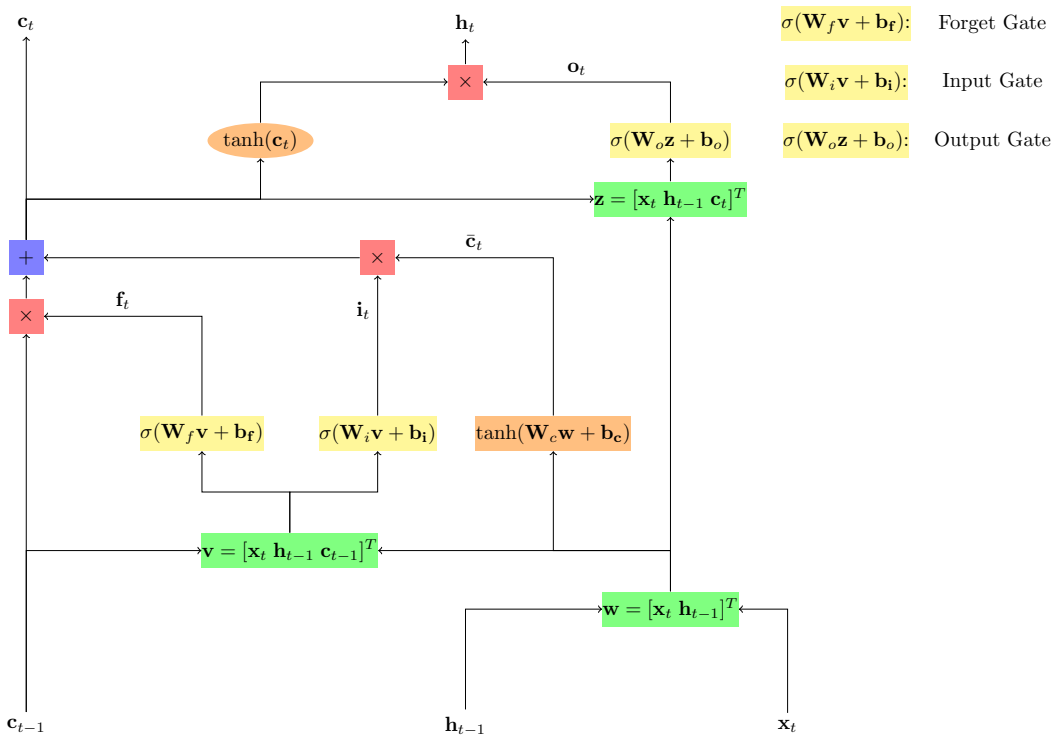


Figure 2.9: Visualization of the LSTM architecture.

Both are labeled $\mathbf{w}$ in figure 2.9, $\mathbf{w}$ contains all information that could possibly be included in the new state. Finally the weighted candidate values are added to what was previously stored. This operation leads to the updated memory state $\mathbf{c}_t$. Last but not least the new output value has to be computed, which will be a filtered version of the cell state. The decision of which and how much of each state variable will be send outside is made by the output gate. It's output $\mathbf{o}_t$ is multiplied with a rescaled version of the cell state. The rescaling is done using another hyperbolic tangent, which again sets all values between minus one and one. The product of this rescaled state and the weights found in $\mathbf{o}_t$ then yields the new output $\mathbf{h}_t$.

At this point it is interesting to note that the graphs considered in this thesis will almost exclusively consist out the exponential, logarithmic, sigmoid and hyperbolic tangent functions. These functions are intimately related. It is well known that the exponential and logarithmic functions are connected by:

$$\exp(\ln(x)) = x \quad \text{if } x > 0. \tag{2.41}$$

The sigmoid and hyperbolic tangent functions in turn consist out of exponentials, and are related trough [12, page 15]:

$$\tanh(x) = 2\sigma(2x) - 1. \tag{2.42}$$

The equation above can be proven by using the definitions of the sigmoidal $\sigma =$

$\frac{1}{1+\exp(-x)}$, and hyperbolic tangent $\tanh = \frac{\exp(2x)-1}{\exp(2x)+1}$:

$$\tanh(x) = \frac{\exp(2x)-1}{\exp(2x)+1} \tag{2.43}$$
$$= \frac{2\exp(2x) - (\exp(2x)+1)}{\exp(2x)+1} \tag{2.44}$$
$$= \frac{2\exp(2x)}{\exp(2x)+1} - \frac{\exp(2x)+1}{\exp(2x+1)} \tag{2.45}$$
$$= \frac{2\exp(2x)}{\exp(2x)+1} - 1 \tag{2.46}$$
$$= \frac{2}{\frac{\exp(2x)+1}{\exp(2x)}} - 1 \tag{2.47}$$
$$= \frac{2}{1+\frac{1}{\exp(2x)}} - 1 \tag{2.48}$$
$$= 2\frac{1}{1+\exp(-2x)} - 1 = 2\sigma(2x) - 1 \tag{2.49}$$

These two non-linear functions are thus very similar, in the LSTM-cell their different function value ranges decide how they are used. Sigmoids determine how much of something should be used, stored, or deleted, because its output values range between zero and one, while computations done on the cell state use the hyperbolic tangent, due to its wider range of function values.

Bidirectional Long Short Term Memory

With the advent of LSTMs deep recurrent networks became feasible in speech recognition [16]. RNNs are always deep in time, because their hidden state depends on past inputs. To enable abstraction their structure must also be deep in space. A bidirectional LSTM layer is shown in figure 2.10. It is important to note, that several LSTM layers are often followed by a single linear layer, as is the case in [16]:

$$\overrightarrow{\mathbf{h}}_t = \text{LSTM}(\mathbf{W}_{\overrightarrow{\mathbf{h}}_t}[\mathbf{x}_t\ \mathbf{h}_{t-1}]^T + \mathbf{b}_{\overrightarrow{\mathbf{h}}_t}) \tag{2.50}$$
$$\overleftarrow{\mathbf{h}}_t = \text{LSTM}(\mathbf{W}_{\overleftarrow{\mathbf{h}}_t}[\mathbf{x}_t\ \mathbf{h}_{t+1}]^T + \mathbf{b}_{\overleftarrow{\mathbf{h}}_t}) \tag{2.51}$$
$$\mathbf{y}_t = \mathbf{W}_y[\overrightarrow{\mathbf{h}}_t\ \overleftarrow{\mathbf{h}}_t]^T + \mathbf{b}_y \tag{2.52}$$

If stacked on top of each other, these bidirectional LSTM layers form a deep recurrent network. Defining $\mathbf{h}^0 = \mathbf{x}$, $\mathbf{h}^N = \mathbf{y}$ looking at time from $t = 1$ to T and taking N layers leads to:

$$\overrightarrow{\mathbf{h}}_t^n = \text{LSTM}(\mathbf{W}^n_{\overrightarrow{\mathbf{h}}_t}[\mathbf{h}_t^{n-1}\ \mathbf{h}_{t-1}^n]^T + \mathbf{b}^n_{\overrightarrow{\mathbf{h}}_t}) \tag{2.53}$$
$$\overleftarrow{\mathbf{h}}_t^n = \text{LSTM}(\mathbf{W}^n_{\overleftarrow{\mathbf{h}}_t}[\mathbf{h}_t^{n-1}\ \mathbf{h}_{t+1}^n]^T + \mathbf{b}^n_{\overleftarrow{\mathbf{h}}_t}) \tag{2.54}$$
$$\mathbf{h}_t^N = \mathbf{W}_y^N[\overrightarrow{\mathbf{h}}_t\ \overleftarrow{\mathbf{h}}_t]^T + \mathbf{b}_y^N \tag{2.55}$$

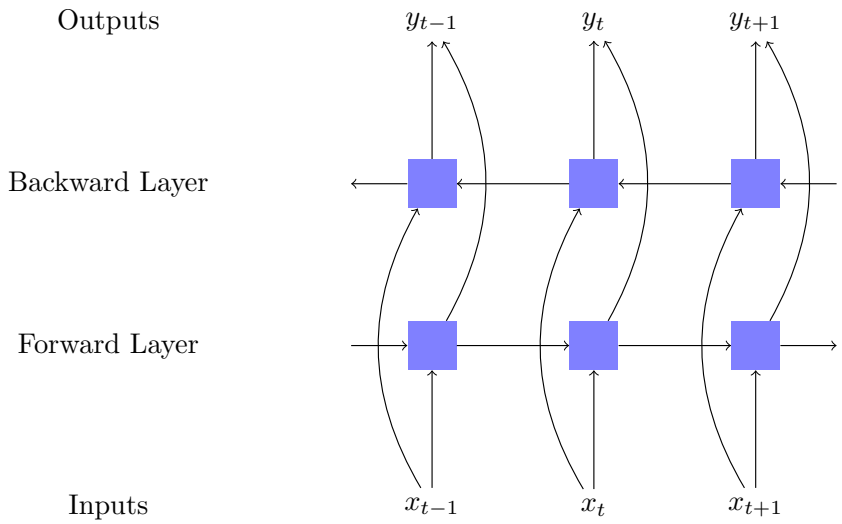


Figure 2.10: A bidirectional Long short term memory layer, according to [16]

In this setting each LSTM cell has access to information from before or after it. For this to work the speech sequence, which is analyzed has to be recorded completely. In this case future information is available and should be used for recognition purposes.

2.3 Connectionist temporal classification

A bidirectional LSTM layer establishes access to past and future input information, but the alignment of the speech inputs and it's transcription remains unknown. Connectionist temporal classification is an output layer, which in tandem with the BLSTM layer enables the network to work around this problem, while at the same time allowing the gradient to propagate trough it. The resulting network can be trained end-to-end.

CTC consists of a softmax layer with one output per alphabet element plus one for blank our no label. The idea is to allow the network to output a label at any input time step, and post process these outputs to get the overall sequence of labels correct, while ignoring the timing of each label. More formally L denotes the used alphabet and $L' = L \cup \oslash$ the alphabet with the empty label. The network output vectors are denoted by $\mathbf{y}^t$ and will have length $|L'|$. Given the training set S and the input sequence $\mathbf{x}$ the probability for observing a sequence π consisting of elements from L' with length T is given by [12, page 56]:

$$p(\pi|\mathbf{x}, S) = \Pi_{t=1}^{T} y_{\pi_t}^t. \tag{2.56}$$

In a first step all a mapping $\mathcal{B}$ is introduced, which simplifies the allowed paths trough the label probabilities, by removing duplicates. For example $\mathcal{B}(x \oslash xy\oslash) = \mathcal{B}(\oslash xx \oslash xyy) = xxy$. In other words a new label is only produced if the output label changed in the path. The probability of a labeling l can be computed by summing

up the probabilities of all paths, which map to the same labeling [12, page 57]:

$$p(\mathbf{l}|\mathbf{x}) = \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{l})} p(\pi|\mathbf{x}). \tag{2.57}$$

Generally speaking this collapsing process of different paths is what makes it possible to use CTC with unaligned data.

Equation 2.57 must be evaluated efficiently, which can be done by using a forward-backward algorithm. Such algorithms compute forward and backward pass variables $\alpha_t(s)$ and $\beta_t(s)$. The two variables break the probability of correctly predicting the sequence into a prefix α and suffix β part. During training their product at any time step t and sequence position s yields the probability sum of all label paths found in the network output $\mathbf{y}$ and mapped onto $\mathbf{t}$ by $\mathcal{B}$. The probability of predicting the targets ($\mathbf{l} = \mathbf{t}$) given the inputs is then given by:

$$p(\mathbf{t}|\mathbf{x}) = \sum_{s=1}^{|\mathbf{t}'|} \alpha_t(s)\beta_t(s) \tag{2.58}$$

Which is running tough the $|\mathbf{t}'|$ elements of the target sequence with added blanks at the start, end and between all labels. The sum checks all possible alignments, which could follow from the mapped outputs. Adding up the negative logarithm of the probabilities above leads to the CTC cost function. Decoding can be done using simlpe greedy decoding or a more complex beam search variant called prefix search. More details can be found in [12, chapter 7], [13, chapter 7] and [15].

CTC takes into account only the features and (depending on how the beam search is done) ignores the label it previously produced. Additionally the number of network outputs is coupled to the input frames. This shortcoming lead to the development of so called transducers. These models decouple the decoding time from the input time and use past labels to improve performance.

2.4 Listen, Attend and Spell [7]

The Listen Attend and Spell architecture is a deep neural network, designed to jointly learn to align and transcribe speech data. In contrast to CTC it belongs to the transducer family of speech recognition algorithms. All transducers have in common that they consist of an encoder and a decoder, with decoupled time steps. The idea also appeared in the automatic translation literature [2]. Working on the source language text, an annotator or encoder produces high level annotations. These annotations are then given to the translator or decoder, which using its own output time step works with the annotations to compute a context, which it then uses to produce a translation. In the speech application the encoder is referred to as the listener. The decoder as the speller. Combined these two form the las-network. The listener is a pyramidal recurrent neural net. It accepts filter bank spectra $\mathbf{x}_n$ as inputs and produces compressed high level output features $\mathbf{h}_m$. Compression reduces the computational load during feature processing later. The speller in turn accepts

the features as input and outputs distributions over character sequences $\mathbf{y}_p$. Due to the uncoupling of the input and decoding time step, it requires a state, as well as an attention mechanism. The state provides a memory of what happened in the decoder in the past. The attention function determines, which listener-features are relevant at a given decoding time step. Combining attention and state makes it possible to label the input data. An overview of the las-achrcitecture is given in figure 2.11.

2.4.1 The Listener

The listener, consists of Long Short Term Memory blocks. These blocks are arranged in layers. The inputs are first fed into a recurrent bidirectional layer (BLSTM). This choice gives the system access to future data, therefore only fully recorded data can be analyzed. This system is restricted to applications, which do not require transcriptions in real time or where its acceptable to wait with decoding until a sentence has been recorded completely. When going up in figure 2.11, pyramidal layers (pBLSTM) follow the initial BLSTM layer. The pyramidal structure concatenates the hidden values computed previously, such that their time dimension is halved:

$$\mathbf{h}_t^n = \text{BLSTM}(\mathbf{h}_{t-1}^n, [\mathbf{h}_{2t}^{n-1}, \mathbf{h}_{2t+1}^{n-1}]) \tag{2.59}$$

Technically instead of two, three or more previously computed feature vectors could be concatenated, which increases the compression factor per pyramidal layer. This operation reduces the length U of the high level features $\mathbf{H}$. Without this compression the following attend and spell operation has a hard time extracting the relevant information, because a longer time span has to be considered to decode a single character. Additionally the compression reduces the problem complexity, which speeds up the training process significantly.

2.4.2 Attend and spell

The speller takes the features and produces a distribution over Latin character sequences as output. The computation of this output involves the context vector $\mathbf{c}_i$, the decoder state $\mathbf{s}_i$, the features $\mathbf{H}$ and the previous output $\mathbf{y}_i$. The index i denotes decoding time, $i-1$ is used to refer to results from the last time step. The last decoding step I, at which the system terminates is a learned quantity. While the last input step U, depends on the input features, lower case u denotes the input step.
During operation the Attend and spell functions keep track of previous output labels and previously important features, which are contained in the context. This information is stored in the state $\mathbf{s}_i$. To function the network must determine, which part of the computed features is $\mathbf{H}$ are relevant at any given decoding time step i. The context vector $\mathbf{c}_i$ contains a linear combination of relevant features, weighted according to their importance. The AttentionContext function determines these weights based on the state. Finally the speller function finds a probability over possible labels using the on the relevant features in the context and the state. The

computing steps are therefore [7, page 4]:

$$s_i = \text{RNN}(\mathbf{s}_{i-1}, \mathbf{y}_{i-1}, \mathbf{c}_{i-1}) \tag{2.60}$$

$$\mathbf{c}_i = \text{AttentionContext}(\mathbf{s}_i, \mathbf{H}) \tag{2.61}$$

$$P(\mathbf{y}_i|\mathbf{x}, \mathbf{y}_{<i}) = \text{CharacterDistribution}(s_i, \mathbf{c}_i) \tag{2.62}$$

The state follows from a recurrent neural a multilayer LSTM. In contrast to the listener the speller is causal, meaning that it makes decisions only based on information computed during previous decoding steps. LSTMs are necessary here, because past states must be remembered. The attention mechanism, called AttentionContext above, computes a new context vector once every time step. This computation starts with the determination of the scalar energy $e_{i,u}$, which will be used as weight for its corresponding feature vector h_u. The computation starts with two feedforward neural networks or multilayer perceptrons (MLP), ϕ and ψ [7, page 5]:

$$e_{i,u} = \phi(\mathbf{s}_i)^T \psi(\mathbf{h_u}) \tag{2.63}$$

$$\alpha_{i,u} = \frac{\exp(e_{i,u})}{\sum_u \exp(e_{i,u})} \tag{2.64}$$

$$\mathbf{c}_i = \sum_u \alpha_{i,u} \mathbf{h}_u \tag{2.65}$$

α is produced by running $\mathbf{e}$ trough a softmax function, which scales $\mathbf{e}$ such that all elements are within $(0, 1)$ and add up to one. These scaled weights, can then be used to form the context vector $\mathbf{c}_i$. When the training process converges the α_is typically follow a distribution with sharp edges[7, page 5]. Thus it is justified to think of the alphas as a sliding window. This window contains only the currently relevant parts of the condensed input data set.

2.4.3 Training

For end-to-end speech recognition all networks must be trained jointly. The objective is to maximize the logarithmic probability:

$$\max_\theta \sum_i \log P(y_i|\mathbf{x}, y_{<i}; \theta). \tag{2.66}$$

Here y_i denotes the current output distribution, x the input, θ the various network parameters and finally $y_{<i}$ the ground truth, which is the known true desired output. In practice the objective is minimized by working with a cross entropy loss function. Using the known output during training creates a situation, where the past outputs are always right. In practice however the situation will be different, as the network is going to make mistakes. As it is desired to create a robust model it is necessary to sometimes include the character distribution generated by the networks being trained. Which leads to the objective [7, page 5]:

$$\hat{\mathbf{y}}_i = \text{CharacterDistribution}(s_i, \mathbf{c}_i) \tag{2.67}$$

$$\max_\theta \sum_i \log P(y_i|\mathbf{x}, \hat{y}_{<i}; \theta) \tag{2.68}$$

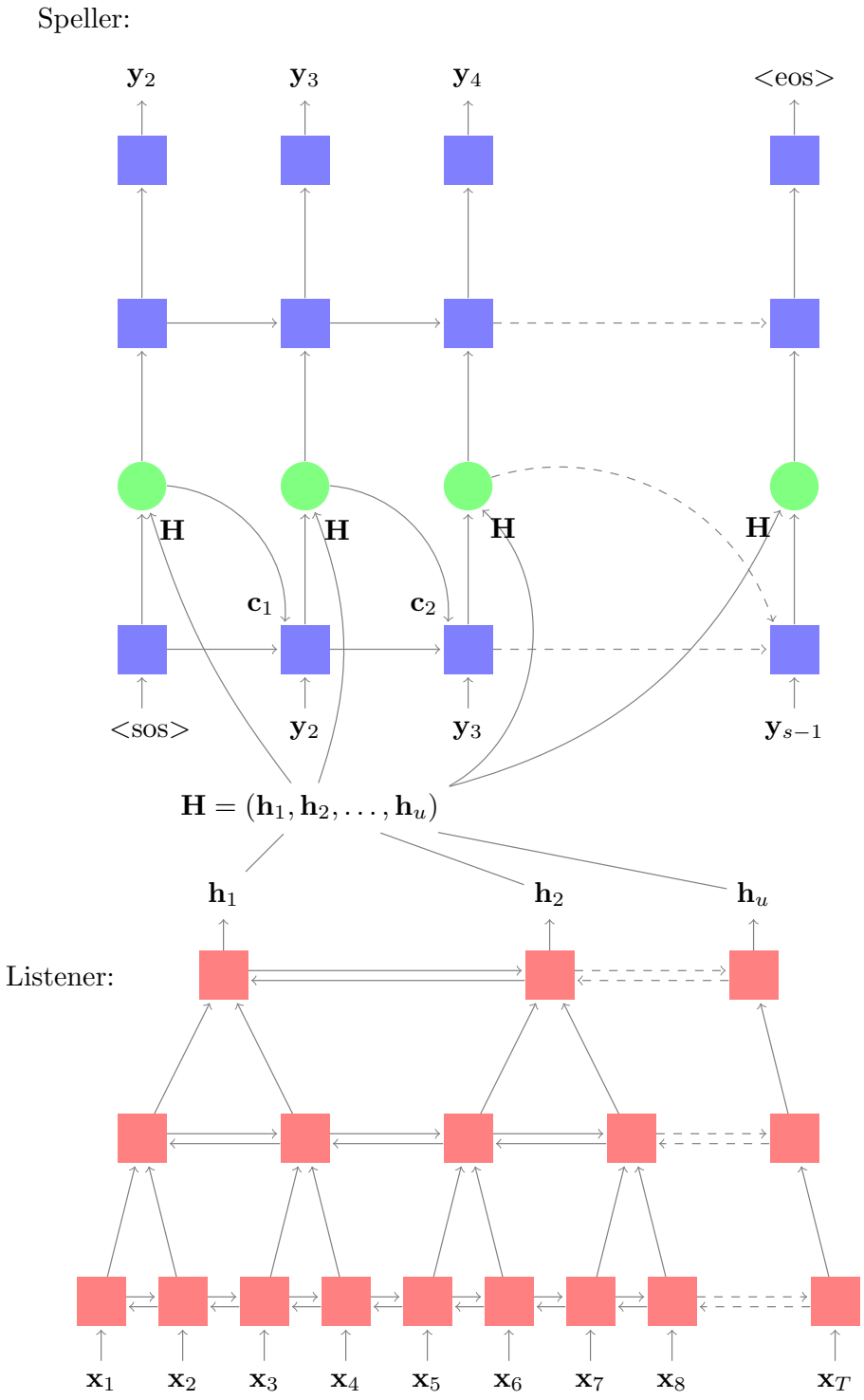


Figure 2.11: The LAS architecture [7, page 3]. BLSTM blocks are shown in red. LSTM blocks in blue and attention nets in green.

The novelty in comparison to the previous expression is that $\hat{y}_{<i}$ is sometimes taken from the past network outputs instead of the ground truth. An idea which Chan et al. found in [4].

2.4.4 Decoding with beam search

In order to generate a readable text, it is necessary to choose characters from the generated character distributions. One way to do this is to simply pick the most likely letter from each distribution. This approach is called greedy decoding. It ignores the possibility of generating better results by also considering less likely options. It is reasonable to expect better results, when considering more then just the most likely label, because the attend and spell decoder takes past labels into account. Consequently a broader search trough the most likely options is considered. Unfortunately memory limitations make it impossible to search trough all possible combinations. Therefore only the n most likely options are explored and the rest is disregarded. This approach is referred to as beam search. Taking into account the most likely options for each label produces a tree of possible transcriptions. The different routes along this tree are called hypotheses. A score for each hypothesis can be computed, by multiplication of the probability values the las-network assigned to each branch along its path. In order to account for different hypotheses lengths the total probability must be divided by the hypothesis length. In beam search only the m most likely hypotheses are kept. Using only las-probabilities is be equivalent taking only the acoustic data and their labels into account. In general text data is far more abundantly available then speech data. To take advantage of these large text corpora a language model trained on these can be used to make a more informed decision when choosing a hypothesis in the beam. A selection can then be made according to [7, page 6]:

$$s(\mathbf{y}|\mathbf{x}) = \frac{\log P(\mathbf{y}|\mathbf{x})}{|\mathbf{y}|_c} + \lambda \log P_{LM}(\mathbf{y}) \tag{2.69}$$

Here P_{LM} denotes the weight the language model assigns to each hypothesis. And λ is a weight factor, which determines the language model importance. The formula above describes beam selection using a language model to re-score the attend and spell probabilities with a language model.

2.4.5 Levenshtein distance

A metric is required to measure the quality is label sequences. The Levenshtein or edit distance is such a metric. It can be thought of as the minimum number insertions, deletions or substitutions required to transform one sequence into another. An example computation for the two words huis and house is shown in table 2.1. The table cells count replacements, deletions or insertions. At position i, j in the table the first sequence is taken into account until element i, likewise the second sequence is considered up to position j. In order to compute the entries in the table

	ϵ	h	u	i	s
ϵ	0	1	2	3	4
h	1	0	1	2	3
o	2	1	1	2	3
u	3	2	1	2	3
s	4	3	2	2	2
e	5	4	3	3	**3**

Table 2.1: An example of the edit distance matrix for the dutch word huis and its english translation house. The edit distance can be found in the cell in the lower right of the table.

the first row and column of the distance matrix D are initialized to:

$$D[0, j] = j \text{ and } D[i, 0] = 0. \tag{2.70}$$

These entries are known beforehand, because if one index is kept at zero, no elements of the corresponding sequence are considered and i or j insertions will always be necessary. If $i \neq 0$ and $j \neq 0$ the following update rule is used [22, slide 10]:

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1, & \text{deletion} \\ D[i, j-1] + 1, & \text{insertion} \\ D[i-1, j-1] + \delta(x[i-1], y[j-1]) & \text{match or replacement} \end{cases} \tag{2.71}$$

$$\text{with } \delta(a, b) = 0 \text{ if } a = b \text{ else } 0 \tag{2.72}$$

Based in these rules the table 2.1, has been computed. Intuitively the distance three makes sense, because to go from huis to house, an o has to be inserted, the i replaced with an s and finally and an extra e has to be added, which amounts to three edits.

2.5 Tensor-flow

In this section is devoted to the toolbox, which will be used to implement the Listen Attend and spell, architecture. According to the Tensor-flow authors [1]:

"TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms".

In tensorflow a directed graph is constructed based on the input operations which define a computational model. The graph represents how data flows trough the model. Each node of the graph has zero or more inputs and zero or more outputs. The nodes represent operations. Tensors flow along the edges of the graph. Graphs are executed in sessions. When running a graph the desired output variables or operations must be specified. Tensorflow then checks if all inputs required to compute the requested output values or perform the desired operation are present. Afterwards its evaluates only the operations including the requested node to find the desired output value or complete the requested operation. Generally tensors do not outlive a single execution of the graph. Variables however are persistent. The model parameters must be

stored in variables. The gradient can then be applied to these variables to improve the model parameters over multiple graph executions.

Chapter 3

Methodology

3.1 Object oriented programming

In computer science object oriented programming is an important paradigm that uses objects to group related data and code into object fields and routines. In addition to better grouping objects allow to encapsulate data, shielding it from access. A good objects will not only encapsulate data but at the same time provide the tools necessary to manipulate the stored data. Ideally future users or other team members will be able to use the object without in depth knowledge of it's internals, which makes team software development easier. An important goal of this thesis project is code re-usability, additionally the work developed should fit nicely into the existing speech recognition toolbox. To allow future users to switch between various algorithms with ease an object oriented design philosophy is followed. This approach enables future users to switch from a CTC based to a listen attend and spell based recognitions system by replacing a CTC graph object with a listen attend and spell graph object.

3.2 Shape invariants

Writing code implementing artificial neural networks is difficult, because the project not likely to work unless all issues in the code have been fixed. A single bug can prevent experiments from working, in such a case the only feedback the programmer gets is a negative experimental result. In order to prevent incorrect implementations of recurrent neural networks it is important to strictly enforce shape invariants of state tensors in loops. This means that the shape of the tensors going into a loop body must have the same size when they leave. This way one can be certain, that for example the network does not change the number of label probabilities it predicts over time. However sometimes the shape of a tensor must be altered, for example if newly computed results become available and a tensors dimension must be increased to create storage space. In such cases shape invariants must be explicitly turned off, that way data can be accumulated over several loop iterations. Shape invariants force the programmer to consciously specify the tensors and the corresponding dimensions

where data accumulate. Thus making these important parts of the code explicitly visible, while at the same time preventing unwanted data accumulation and memory consumption elsewhere.

3.3 Code quality

When writing code it is important to ensure that not only the author, but also future readers and coworkers are able to understand it. An important part of ensuring future usability is to document functionality. However readable code is not just well documented, but also respects well established coding conventions. An important collection of python code conventions is the python software foundation's PEP 8 style guide [28]. However following these conventions is tedious and easily forgotten. Additionally software development is team work and the best way to ruin the esprit de corps is an environment where team members constantly complain to each other about code convention violations. A good way to enforce decent coding style is to get the computers to help. In the python world a useful tool for this purpose is `pylint`[1]. `Pylint` is a code analysis tool that looks for patterns in the code that might indicate bad style, for example incorrect variable names, missing documentation, wrong indentation and so forth. Being an interpreted language python code does not need to be compiled. This advantage comes with an important drawback, because during compiling many errors in the code are found and reported to the programmer. Python can do the same thing during runtime, but it crashes when an error is encountered and already computed data is often lost during those crashes. Pylint therefore taps into the python interpreter to scan the code for errors like a compiler. By alerting the programmer to potential issues in the code before execution, pylint can decrease the number of runtime crashes and increase development efficiency.

3.4 Version control

Version control software tracks changes to the codebase under development and allows authors to back up versions of the code as revisions. Git[2] is a popular version control tool. Git keeps all past revisions in a graph. The full graph is a recording of the projects history since its inception. However version control is more than simply keeping a record of past changes. Depending on the project adding new features bears the risk of breaking older code. If things go badly the code can end up in a state, where neither the old nor the desired new functionalities of the code work. In order to always maintain a working core of the code, version control software should be used to split of the development of new functionalities into branches. Should serious problems arise, which where not anticipated when the new feature was planned, the new branch can simply be discard, leaving the core code intact. Should the new functionality work, it's branch is merged into the core code, which remained fully functional at all times. Additionally working with branches facilitates

[1] https://pylint.readthedocs.io/en/latest/intro.html
[2] `https://git-scm.com/`

software development in a Team, as different members can work independently on their own branches. Without branching working with several people on the same files can become tedious, if conflicting changes are made. Branched work lets team members focus on development first, potential conflicts are then resolved later, when the various tree branches are merged.

Chapter 4

Implementation

This implementation chapter covers key architecture aspects and the reasoning behind design decisions made while implementing the speech recognition system.

4.1 Loading data and feature extraction

During system training as well as decoding usage at later stages, an infrastructure must be in place to load and process the input data. In the training phase the target data must be loaded, normalized, encoded and made available to the optimization algorithm in the correct form. The shape and encoding of the targets depends on the algorithm used. Connectionist temporal classification requires a blank output label for example, which listen attend and spell does not need. The code written to handle the data preprocessing fulfills these requirements.[1]

4.2 Efficient sequence to sequence implementations

When working with sequence to sequence methods such as CTC, data processing problems arise naturally. For efficiency reasons it is beneficial to work with fixed size data tensors. However the number of frames varies significantly between utterances. The same holds true for the target lengths. In order to still be able to work with fixed size tensors, the inputs and targets must be padded with zeros or unknown tokens to fill up the length differences. In order to keep the speed benefit of working with tensors it is important to keep track of the sequence lengths of every individual utterance. The sequence length information must be used to avoid actually processing any of the padded data, that the tensors where filled up with in order to allocate memory faster. The listener produces a three dimensional logit tensor of size $[B, T, F]$, where B denotes the batch size, T the largest occuring frame number and F the feature dimension size. When implementing the output layer the naïve approach would be to loop over the time dimension, and evaluate the linear layer T times with

[1] Its development history as well as the process of harminizing the interfaces of the thesis code with the rest of the toolbox is recorded in the project git branch at `https://github.com/vrenkens/tfkaldi/commits/las`.

a B times F matrix. Which is simple to implement but very inefficient. Instead the sequence information should be used to remove the padding and concatenate the batch dimension into a matrix $\mathbf{X}$ of size $[S, F]$, where S denotes the sum of all sequence lengths. The linear layer can then be evaluated as:

$$\mathbf{WX} + \mathbf{b} = \mathbf{R} \tag{4.1}$$

The result matrix R can then be converted back into a padded tensor using the sequence length information one more time. An alternative approach which avoids reshaping is to implement a linear recurrent network cell with a state size of zero and use dynamic unrollings the evaluate the linear layer. The resulting memory requirements and speed are comparable.

4.3 Design of the BLSTM-CTC model

Sequence labeling using confectionist temporal classification happens in two phases. First BLSTM layers compute annotations based in the input vectors. These annotations are then handed to a CTC layer which runs the annotations trough a softmax to produce normalized label probabilities and computes the loss during training or searches trough the probability distributions during decoding. The BLSTM-CTC implementation reflects this procedure by splitting the BLSTM layers as well as the training and decoding code into model, trainer and decoding classes. [2]

4.4 Implementing Listen attend and Spell

The listen operations computing the high level feature matrix $\mathbf{H}$ can be completed independently before the attend and spell code is run. All listening related code has therefore been grouped in a Listener class, all attend and spell related functions are grouped in a speller class.

4.4.1 The Listener

The Listener consists of an initial bidirectional long short term memory layer (BLSTM), followed by pyramidal LSTM (PLSTM). The plstm layer compresses the time dimension as described in chapter 2.4.1. The compression has been implemented by looping trough the hidden output of the previous layer in time and concatenating every second vector with the one before it. Every PLSTM layer therefore halves the time dimension and produces an output of two times the state size of it's LSTM cells. For use with CTC a liner output layer can optionally be added to the listener, which maps the output from two times the state size to the number of required labels.

[2] It was not necessary to implement CTC decoding. Tested code is already available at `www.tensorflow.org/api_docs/python/nn/connectionist_temporal_classification__ctc_`

4.4.2 The Speller

Based on the high level features **H** computed by the listener the speller evaluates an internal attention mechanism. This mechanism which can be interpreted as a learned sliding window. The speller learns where to place this window based on the decoder state $\mathbf{s}_i$. This state therefore represents a query, which asks the attention mechanism to provide certain information. This query in turn is computed by the spellers internal RNN. Using the attention factors a context vector **c** is computed, which together with the state is used to assign a label. The attention computations described above and in more detail in 2.4.2, do only depend on data available at the current time and previously computed values. The spellers key functions can therefore be implemented within an RNN cell. The state of this augmented RNN cell at decoding time step i is given by:

$$\text{state} = [\mathbf{y}_{i-1}, \mathbf{s}_{i-1}, \mathbf{c}_{i-1}] \tag{4.2}$$

The one hot encoded previous label is denoted by **y** the decoder state by **s** and context by **c**. A visualization of the attend and spell cell implementation is shown in figure 4.1. The blue box surrounding the attend and spell represents a `while` or `for` loop. The loop type depends on whether the speller is run in training or decoding mode. The ground-truth labels are known during training. The attend and spell cell must therefore be evaluated until a sequence of the same length as the ground-truth has been obtained. Taking knowledge of the target sequence length into account the unrolling can be done in a `for` loop. Running in decoding mode is harder, because the length of the sequence the model will assign to the input is unknown.

Design of the greedy decoding loop logic

The attend and spell cell must be evaluated until it produces an end of sentence token or an iteration maximum has been reached. The system should be able to decode multiple sequences in parallel. In order to keep track of the active sequences a done-vector **d** is introduced. **d** contains one entry per sequence, each entry should contain `True` if an `<eos>` token has been produced during decoding of the corresponding input and `False` if not. To achieve the desired behavior the following loop logic has been devised:

```
while keep_working:
        not_done_count = reduce_sum( logical_not( d ))
        done = equal(not_done_count, 0)
        stop_loop = logical_or(done, greater(time, max_steps))
        keep_working = logical_not(stop_loop)
```

The pseudocode above uses relies on the mask **d** and an upper step limit to check if the while loop should continue running. In order to implement decoding efficiently inside of a graph it is important to keep track of the sequence lengths. The decoding loop implementation does this when it determines the done-mask **d**. During every iteration the loop body updates the sequence lengths with the current decoding time, unless **d** contains `True` at the position corresponding to the current utterance.

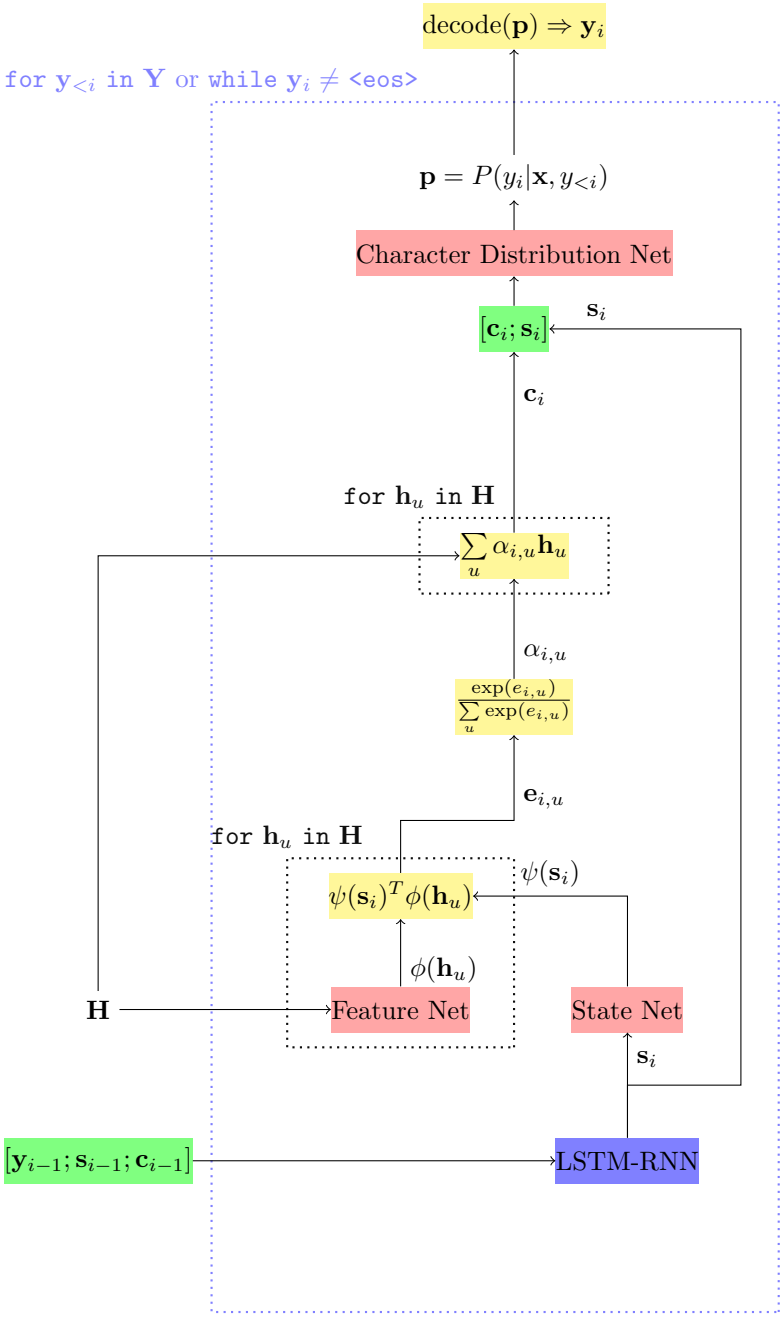


Figure 4.1: Schematic of the attend and spell cell components.

```
decoded = decode(logits)
mask = tf.equal(decoded, <eos>)

time_vec = ones(self.batch_size)*(time+1)
sequence_lengths = select(d,
                          logits_sequence_length,
                          time_vec)
d = logical_or(mask, d)
```

The code listing above updates **d** after the sequence lengths have been updated, because the `<eos>` token should be recorded as well. It is assumed that the decoded function returns a one hot vector and `<eos>` represents an encoded end of sentence token. **d** is initialized to `False` for all sequences.

Design of the beam search decoding loop

The author is unaware of a debugging tool, which would allow to stop executing a tensorflow graph at a given point and examine the graph variables. The beam search operations have therefore been designed using print statements only. In order to reduce the workload the loop logic has been kept exactly the same as the one used for greedy decoding. Instead of processing multiple utterances at the same time, beam width different hypotheses of the same utterance are dealt with. Decoding of this single sample stops, if all beam elements produced an end of sentence token. Furthermore the beam search code keeps the labels, probabilities, cell-states, sequence lengths and done-masks in sorted order for the entire beam.

Cell efficiency improvements

In figure 4.1 the feature net output $\psi(\mathbf{h_u})$ is evaluated inside the cell. While closely following the equations in [7], evaluating the feature net inside the cell means accessing $\mathbf{H}$ and evaluating an MLP at every decoding time step. Instead the $\psi(\mathbf{h_u})$ can be computed outside of the cell ahead of decoding time and a matrix $\hat{\mathbf{H}}$ consisting of $\psi(\mathbf{h_u})$ for all u is used inside the cell. Furthermore all computations have been implemented using matrix tensor multiplication functions instead of for loops, which yields large performance increases. Figure 4.2 shows a tensorboard[3] visualization of the attention computation. The rose colored blocks represent matrix multiplications.

[3] `https://www.tensorflow.org/versions/master/how_tos/graph_viz/`

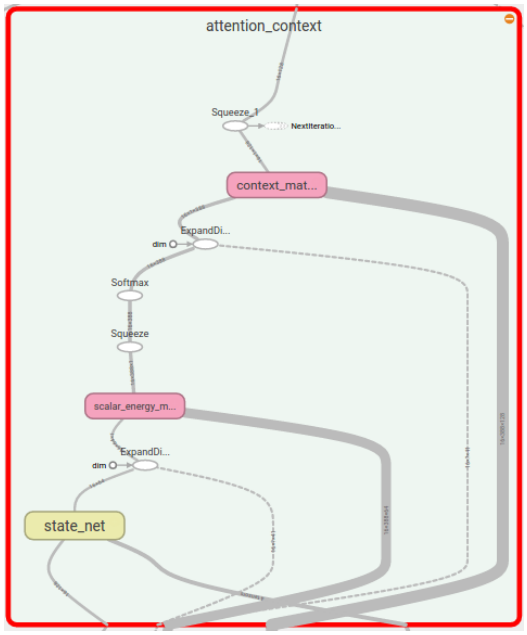


Figure 4.2: Tensorboard visialization of the attention context computations.

Chapter 5

Experiments

5.1 TIMIT

The *timit* speech corpus [10], contains recordings of ten phonetically rich sentences, for example:

> "She had your dark suit in greasy wash water all year. "

For each sentence a transcription of the spoken phonemes is also available. Phonemes are sets of sounds, which are considered equivalent in a given language. In alphabetic writing systems such as the latin one the phoneme to letter mappings should ideally be one. Due to the fact that the Latin script was devised for classical Latin as well as the fact that when pronunciation changes the spelling often remains the same, the phoneme to letter mappings are often far from one. Therefore the `timit` data set comes with phonetic transcriptions for all sentences. For the sentence considered above the spoken phonemes are:

> "h# sh ix hv eh dcl jh ih dcl d ah kcl k s ux q en gcl g r ix s ix w ao
> sh epi w ao dx axr ao l y ih axr h# "

The transcriptions contain a total of 64 possible phonetic labels, in the literature the full set and foldings with 48 and 39 labels are considered [23]. In all following experiments the 39 labels shown in 5.1 will be considered.

For all experiments the timit data set is split into a training, validation and test set. Containing 3696, 400 and 192 sentences [13, page 80].

5.2 BLSTM-CTC

This section explores bidirectional long short term memory layers with CTC output on the timit corpus [13, 15]. Training began after transforming of the speech data into Mel frequency cepstral coefficients (*MFCCC*s) augmented with first and second derivative information. The raw time domain signal as well as the feature vectors are shown in figure 5.2. The phonemes where folded as described above. Two *BLSTM* layers where stacked on top of each other, both using LSTM cells with a

TABLE I
LIST OF THE PHONES USED IN OUR PHONE RECOGNITION TASK

Phone	Example	Folded	Phone	Example	Folded
iy	*beat*		en	*button*	
ih	*bit*		ng	*sing*	eng
eh	*bet*		ch	*church*	
ae	*bat*		jh	*judge*	
ix	*roses*		dh	*they*	
ax	*the*		b	*bob*	
ah	*butt*		d	*dad*	
uw	*boot*	ux	dx	(butter)	
uh	*book*		g	*gag*	
ao	*about*		p	*pop*	
aa	*cot*		t	*tot*	
ey	*bait*		k	*kick*	
ay	*bite*		z	*zoo*	
oy	*boy*		zh	*measure*	
aw	*bough*		v	*very*	
ow	*boat*		f	*fief*	
l	*led*		th	*thief*	
el	*bottle*		s	*sis*	
r	*red*		sh	*shoe*	
y	*yet*		hh	*hay*	hv
w	*wet*		cl (sil)	(unvoiced closure)	pcl,tcl,kcl,qcl
er	*bird*	axr	vcl (sil)	(voiced closure)	bcl,dcl,gcl
m	*mom*	em	epi (sil)	(epinthetic closure)	
n	*non*	nx	sil	(silence)	h#,#h,pau

Figure 5.1: 48 to 39 phoneme folding as shown in [23].

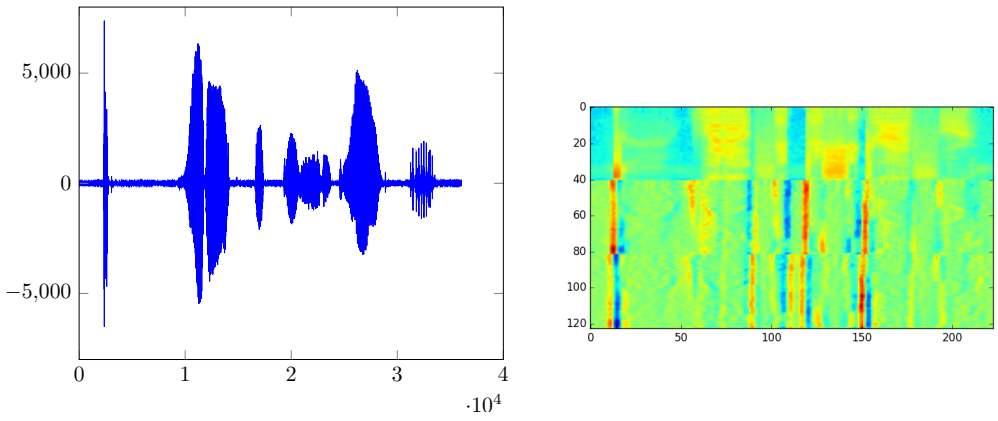


Figure 5.2: Raw time domain signal and feature vectors. Features ecnlude 40 mel banks as well as the first and second derivatives. Raw data was taken from TIMIT utterance faem0-si2022 “What outfit does she drive for?”.

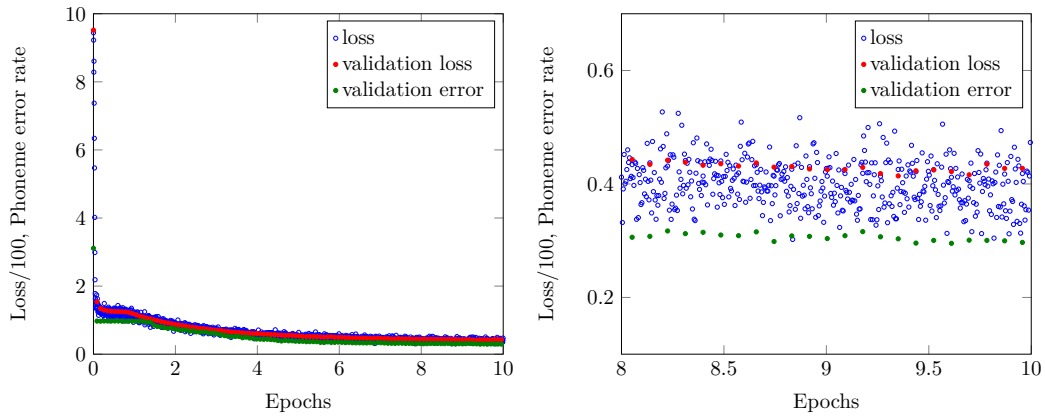


Figure 5.3: Validation and Test set error while training a two layer BLSTM network with CTC output layer on the TIMIT speech corpus with 39 folded phonemes.

dimension of 64 in each direction. The logits computed by the two layers where then fed into a CTC output layer, finally beam search decoding with a beam width of 100 was used for find the phoneme predictions. The whole network was optimized using gradient descent and a learning rate of 10^{-3}. To ensure generalization normal input noise $\mathcal{N}(\mu = 0, \sigma = 0.65)$ was added to the inputs. All feed-forward weights where initialized using another Gaussian $\mathcal{N}(0, 0.1)$, lstm weights were initialized using a random uniform distribution $\mathcal{U}(-0.1, 0.1)$. Gradients where clipped such that all gradient elements are within $(-1, 1)$. Results of the training process are shown in figure 5.3. In the experiment the training process was stopped after 10 epochs to be able to compare the result to the pyramidal listener experiment in the next chapter, at this point the phoneme error rate of this layout was at 29%. However if the training process is continued with decreasing learning rates the error rate values end up close to 26% eventually. Performance can be increased further by assigning the LSTM cells a larger dimension.

5.3 Listen attend and spell experiments

In this section experiments use the full listen attend and spell architecture. After verification of the listener using a CTC output layer, the listen attend and spell network is first tested using greedy decoding. In contrast to beam search greedy decoding, does not maintain several hypotheses, instead it works with the most likely label each time step.

5.3.1 Testing the Listener

A crucial part of the listen attend and spell architecture is formed by the listener. Before working with a fully-fledged las a CTC-layer will be attached to the listener. The idea is to verify the implementation. If CTC can extract relevant information from the listener, the attend and spell code should be able to do the same in later

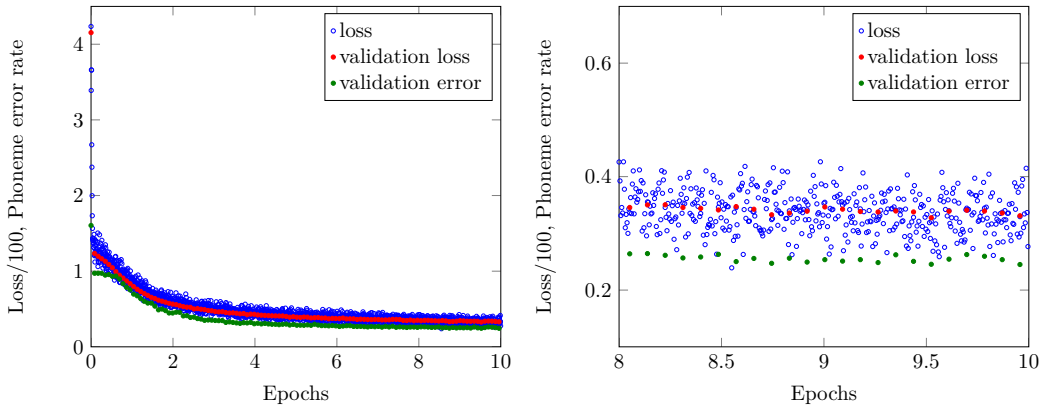


Figure 5.4: The training progress shown for the Listener with added CTC layer. The loss values show in blue and green have been scaled with $\frac{1}{100}$. On the right a closeup on the last two training process is shown.

experiments. In order to keep memory requirements manageable in later experiments with the same listener all LSTM cells where set up with 64 hidden units. As the listener layers are bidirectional this means 64 in each direction so the hidden dimension is 128 in total. This sum is important, because the feature dimension of the lstm outputs is concatenated for each time step. If not further action is taken the listener produces features with a dimension of two times the number of elements per lstm. CTC runs the logits it is given trough a softmax layer to compute label probabilities. To function it must therefore be given a logit tensor, where the feature dimension is equal to the number of labels, the system is supposed to output. To meet this requirement an extra linear output layer has been added to the listener which maps the feature dimension to 40, as required. Figure 5.4, shows the optimization algorithms progress, as measured by trainig loss, validation loss and validation set decoding phoneme error rate. The training was stopped, when the decoding results where no longer improving. During testing a phoneme error rate of `0.268` was observed, which is a pretty good results compared to the twenty four percent error rate of a comparable full BLSTM architecture given that the pyramidal layer compressed the time dimension into half of its original size. During decoding the CTC beam width was once more set to 100.

5.3.2 Greedy Decoding Experiments

Using the tested listener with 64 hidden lstm units per direction and one pyramidal layer, the CTC layer is replaced with attend and spell functions. For computational efficiency these functions have been implemented within a customized RNN cell. Using an RNN framework makes it possible to use optimized code to unroll the attend and spell computations. Among other things dynamically unrolling the second part of the network this way ensures efficient memory utilization and sequence length management. Within the new cell the decoder state size was chosen to be 128,

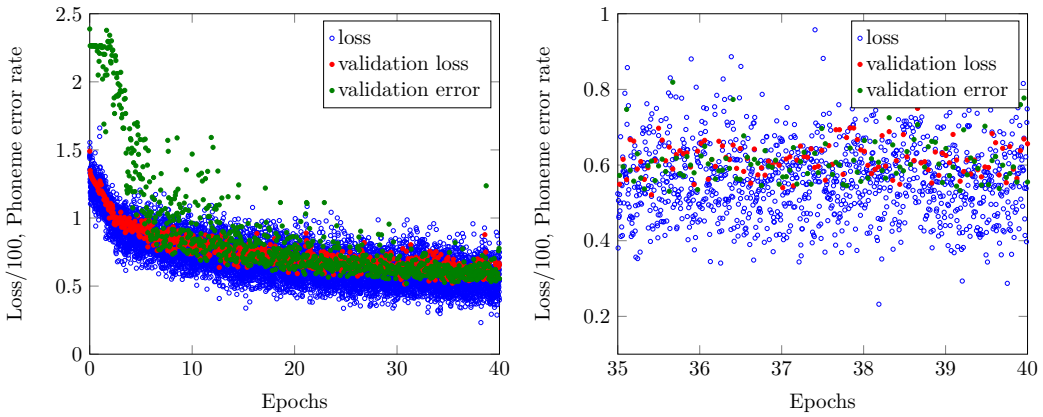


Figure 5.5: The training progress shown for the full las architecture with greedy decoding. The loss values show in blue and green have been scaled with $\frac{1}{100}$. On the right a closeup on the last two training process is shown.

considering the fact that the listener outputs features of size 128, which in turn determines the context vectors to have this same dimension. Hoping to provide sufficient memory to remember past context vectors the decoder state RNN was set to the same dimension. The state and feature networks, ϕ and ψ were given one hidden layer each, with a hidden dimension of 64. This choice was made mainly to conserve memory. During training the network output was used instead of the true target with a probability of 0.7. Figure 5.5 shows an overview of the training process. When considering the last five epochs, the decoding error ranges between 0.5 and 0.9. This means that in the best case half of the labels produced by the system must be modified in order to get to the target sequence. Considering timit utterance `fmld0_sx295` the folded transcription with additional start and end of sentence tokens is given by:

Listing 5.1: Targets

```
<sos>  sil  ih  f  sil  k  eh  r  l  sil  k  ah  m  z
       sil  t  ah  m  aa  r  ah  hh  ae  v  er  r  ey
       n  jh  f  er  m  iy  dx  iy  ng  ih  sil  t  uw  sil
<eos>
```

From the input features the las network decodes:

Listing 5.2: Network output

```
<sos>  sil  hh  ih  f  sil  k  er  r  ow  ow  sil  sil
       t  ah  m  aa  hh  hh  ae  v  er  r  r  n  n  sil
       f  er  er  m  iy  iy  iy  iy  iy  iy  iy  iy  sil
       sil  t  uw  sil
<eos>
```

The decoding and target sequences clearly bead some resemblance. However significant errors do exist in the network output in particular in the last third. The

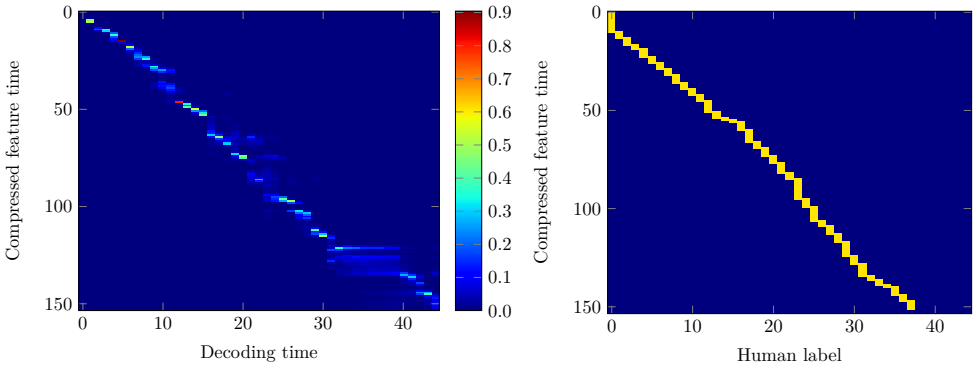


Figure 5.6: Plot of the alignment vectors computed by the network for all 45 labels assigned to timit utterance `fmld0_sx295` (left), and alignments assigned by a human listener (right).

phoneme sequence dx iy ng ih is incorrectly transcribed as iy iy iy iy iy iy iy iy, which has a large impact on the error. The levenstein distance between the two labellings is 21. Given that the target sequence contains 45 labels including the start and end of sentence tokens, the error rate of this example is 0.5, it is therefore slightly better than the average of ≈ 0.55, which is the average decoding error rate over the entire validation set.

5.3.3 The attention mechanism

In order to further verify the implementation of the speller all attention weights α (equation 2.65), have been extracted during decoding. The left side of figure 5.6 shows these vectors in concatenated matrix form. The speller assigns 45 labels. During each decoding time step one label is produced, therefore the attention matrix has the same number of columns. The y-Axis shows compressed input time. This dimension depends on the length of the input utterance under consideration and the compression level, which is set by the number of pyramidal LSTM layers. The attention weights are normalized using a softmax function, hence all elements must sum to one. The scale shown next to figure 5.6 therefore ends at one, values close to one indicate that the network is focusing on a single feature. Generally the attention weights should gradually cover all relevant features over time, this is definitely the case, as the attention weight is clustered around the matrix diagonal. Furthermore only a limited number of compressed frames can be relevant at any given time, ideally each vector should contain a sharp peak at the most relevant frame. The timit data corpus comes with alignments found by human listeners. The recoding of utterance `fmld0_sx295` lasts for $3.018s$. During the feature computation one frame is placed every $0.01s$. Therefore this utterance will come with 302 frames. Using a listener with a single pyramidal layer 151 high level feature vectors will be computed for this utterance. The right side of figure 5.6 shows the rescaled human alignment. The two plots show some resemblance, which indicates that the implemented attention

mechanism is working properly. With the parameters set as described the in the previous section the attention mechanism breaks down towards the end, which can be seen in the closeup shown in figure 5.9. When the speller produces the incorrect sequence of iys the attention weights are spread out over roughly 15 feature vectors.

5.3.4 The effect of the groundtruth selection probability during training

In the previous experiment the groundtruth was only used instead of the network output in 0.3 percent of the cases. As the network output is often incorrect during training, less emphasis will be put on past outputs later during decoding. This section investigates other values. The same las network is retrained for 10 epochs using output probabilities of $0.2, 0.4, 0.6, 0.8$. Results are shown in figure 5.10. It can be observed, that decoding results improve when reduced emphasis is placed on past labels during training. This observation is confirmed by the phoneme error rates of $2.08, 1.97, 1.1168, 0.87$, which where observed on the validation set. To explore the effect of a low reuse probability over longer training periods the experiment has been repeated one more time with a 0.5 groundtruth probability over 40 epochs. Figure 5.11 depicts the training process. In comparison to the experiment show in figure 5.5 the network does significantly better during training, but this improvement does not translate into a better decoding performance.

Ideally one would like to observe the opposite. If the output labels where correct more often during decoding, the networks trained to rely on them should outperform those which where not. Based on the observations above we conclude that beam search should be implemented in order to obtain labels of higher quality during decoding, which in turn should be beneficial to networks, which learned to rely on past outputs.

5.3.5 A second attend and spell cell type

Looking at figure 2.11 it is not perfectly clear weather the blocks after the context computations represent LSTM-cell or are simply part of the feedforward labeling network. Until now it has been assumed that no additional recurrent LSTM network after the context is part of LAS. In this section such an extra RNN will be investigated briefly. The attend and spell cell variation is shown in figure 5.12. The state of the extra memory cell is labeled with $\mathbf{d}$. A learning curve over 10 epochs using a ground-truth probability of 0.3 with greedy decoding is shown in figure 5.13. The results are slightly worse then what was observed with the original setup, despite the extra weights. This result indicated that the extra post context RNN has no additional value. It can therefore be concluded that the decoder state $\mathbf{s}_i$ is sufficient to remember past context information.

5.3.6 Beam-search and Dropout

Two measures have been taken to increase system performance. First a beam of las labels and states is kept, in order to explore several labeling hypotheses as described

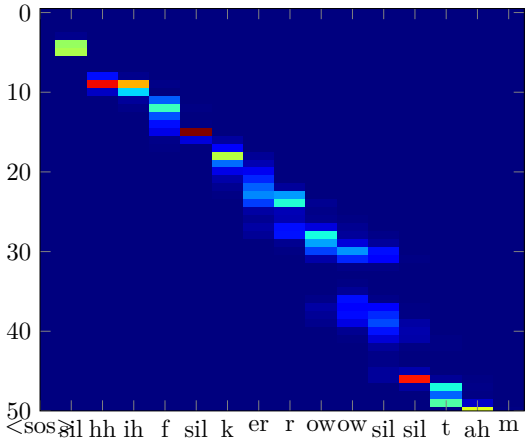


Figure 5.7: Closeup on interesting parts of the first third of the attention matrix.

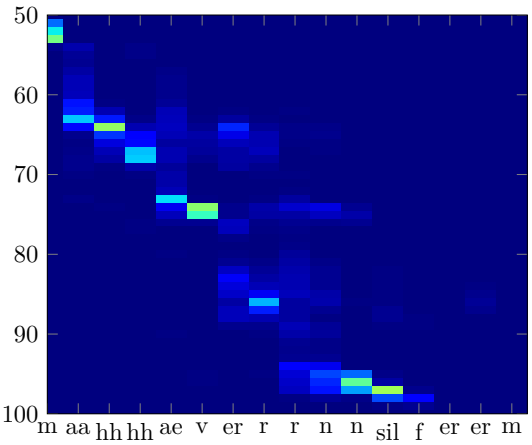


Figure 5.8: Closeup on interesting parts of the second third of the attention matrix.

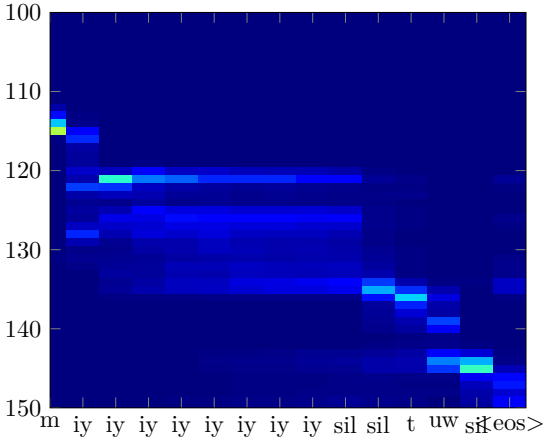


Figure 5.9: Closeup on interesting parts of the last third of the attention matrix.

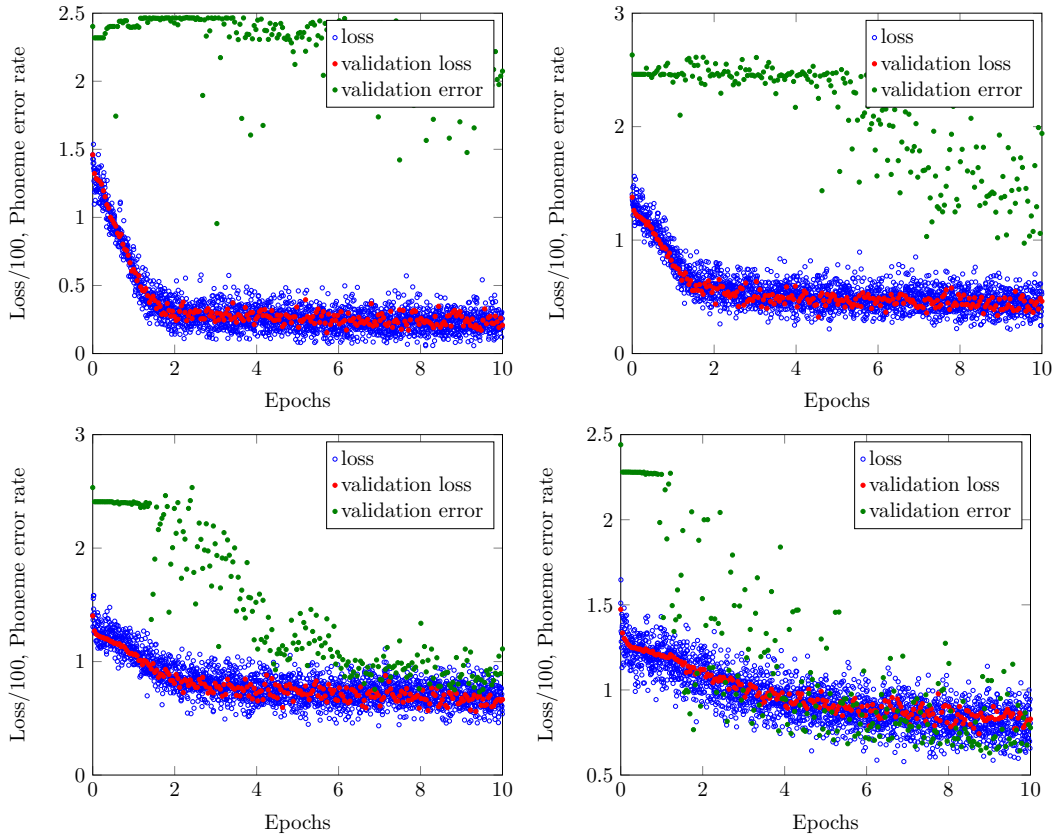


Figure 5.10: Repetitions of the same experiment with network output reuse probabilities $0.2, 0.4, 0.6, 0.8$.

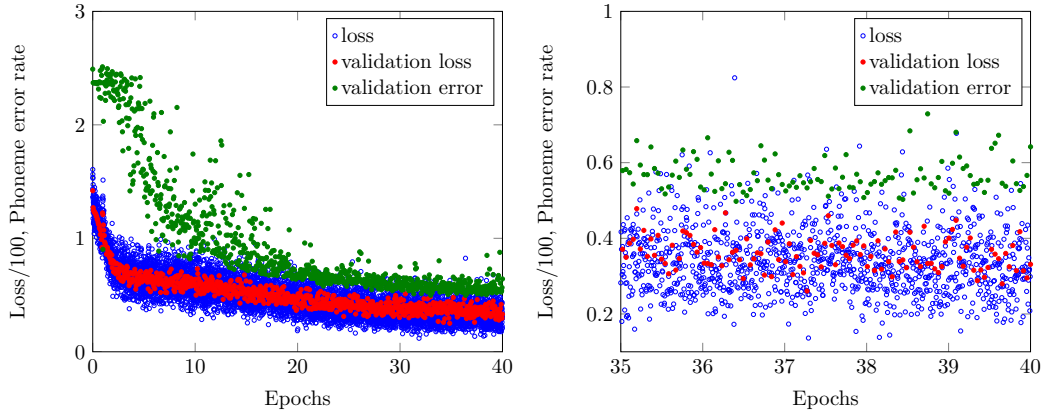


Figure 5.11: Training full las over 40 epochs with a network output reuse probability of 0.5

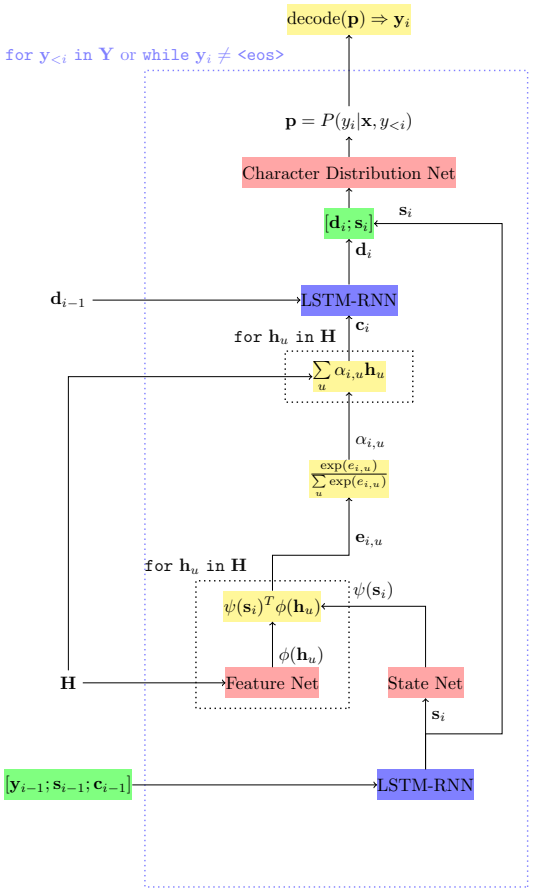


Figure 5.12: A different attend and spell cell configuration, featuring an additional post context RNN

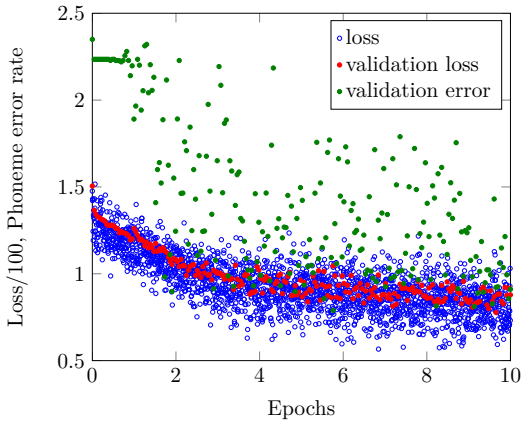


Figure 5.13: Learning curves over 10 epochs using greedy decoding and a training network output reuse probability of 0.7.

in chapter 2.4.4. However the current version of the code does not use language model rescoring.

Second dropout has been added as shown in figure 5.14. The rationale behind using dropout regularization is to allow larger networks to be trained longer without running into over-fitting problems. In comparison to the previous experiments all network parameters have been doubled. Which means that all BLSTM layers used 128 units, all LSTM layers used 265 and all feedforward layers again 128 units. Due to good results in previous experiments the network output reuse probability during training was kept at 0.6. The settings described above led to the learning curves shown in figure 5.15. The validation error drops consistently under 50 percent, without large outliers. Considering once more utterance `fmld0_sx295`.

Listing 5.3: Targets

```
<sos>  sil  ih  f  sil  k  eh  r  l  sil  k  ah  m  z
       sil  t  ah  m  aa  r  ah  hh  ae  v  er  r  ey
       n  jh  f  er  m  iy  dx  iy  ng  ih
       sil  t  uw  sil
<eos>
```

The network now decodes:

Listing 5.4: Network output

```
<sos> sil hh ih f sil k ih r ow sil k ah m sil
          sil t ah m aa aa hh hh v v er ey
          n n sil f f er m iy iy iy iy sil
          sil t uw sil sil
<eos>
```

Which is considerably better than the result obtained earlier. The levenstein distance over target length ratio drops to 0.36 for this utterance. However over the entire validation set 0.45 has been measured. This result is considerably better than the 0.55 which where observed in section 5.3.2 using a much simpler model with greedy decoding and input noise regularization.

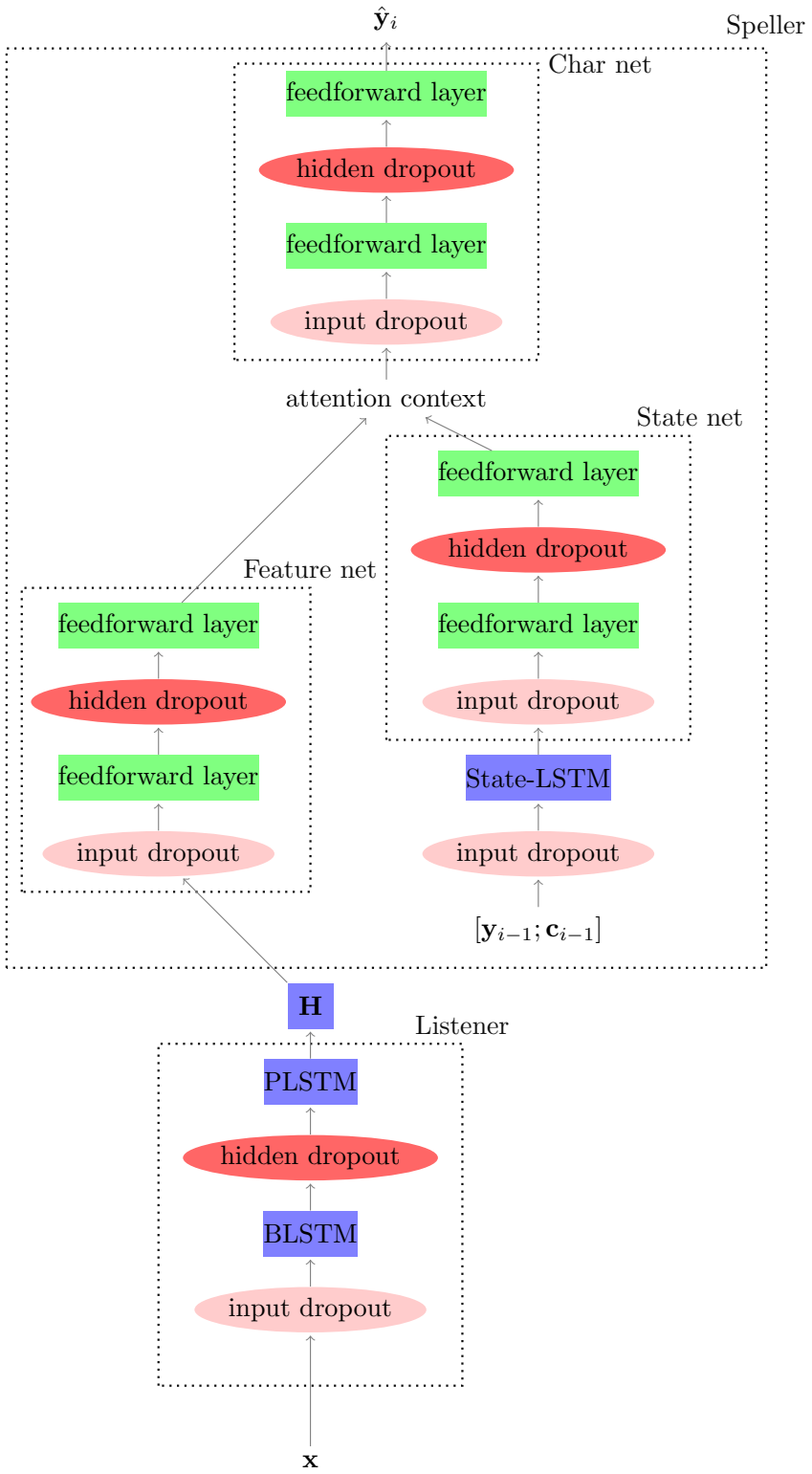


Figure 5.14: Listen attend and spell forward connections with added dropout. No recurrences are shown.

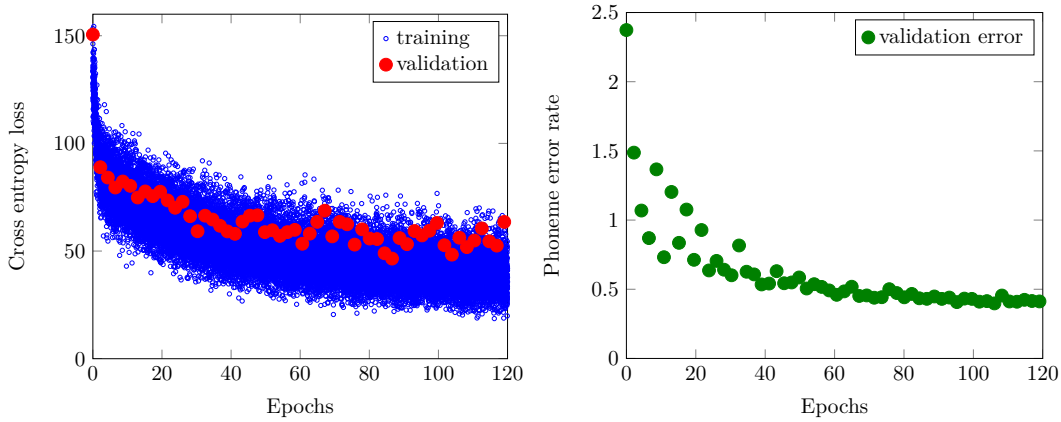


Figure 5.15: LAS learning curves with a listener LSTM using 128 units per direction and one PLSTM. The speller LSTM state size was set to 256. All feed-forward networks in the speller had 128 units per layer and two layers overall.

Chapter 6

Conclusion

In chapter 1 important questions where asked, which this thesis attempts to answer. Based on the experimental evidence gathered it can be said that the implemented listen attend and spell model is able to recognize speech and determine relevant parts of input recordings. The attention weights computed by the model as well as the actual labeling do resemble results obtained by human listeners. This result suggests that [7] correctly claims that neural attention mechanisms can be used to align text to speech data. In terms of network regularization dropout [27] has been found to be a useful tool. Working network parameters turned out to be scaled down versions of the ones stated in the original las paper [7]. for the feedforward parameters which are not mentioned it has been found that 128 units and one hidden layer leads to convergence on timit. However the author is confident that further tuning and exploring deeper configurations will lead to additional improvements.

The goals of this thesis project where to implement a skeleton las model, beam search decoding, and explore different regularization models if possible. Experimental evidence has been gathered, which suggest that the all three goals have indeed been achieved. Some further experimental validation is required. In particular additional separate testing of the beam search and dropout code, which has been tested together in order to satisfy time constraints.

Another important part of this thesis, was to improve the authors insight into software development methods. The thesis deliverables include more than thousand lines of documented and pylinted python code. The code fully integrates into the existing toolbox under development in the speech group at esat. The author hopes, that this work will serve as a good foundation for future development.

Bibliography

[1] A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.0, 2016.

[2] D. Bahdanau, K. Cho, and Y. Bengio. NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE. *CoRR*, abs/1409.0:1–15, 2014.

[3] S. Ben-david. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge Cambridge university press, Jerusalem, 2014.

[4] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. *arXiv*, pages 1–9, 2015.

[5] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. *IEEE International Conference on Neural Networks - Conference Proceedings*, 1993-Janua:1183–1188, 1993.

[6] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[7] W. Chan, N. Jaitly, Q. V. Le, and O. Vinyals. Listen, attend and spell. *arXiv preprint*, pages 1–16, 2015.

[8] Christopher Olah. Understanding LSTM Networks, 2015.

[9] M. Diehl. Script for Numerical Optimization Course B-KUL-H03E3A. Technical report, Optimization in Engineering Center (OPTEC) and Electrical Engineering Department (ESAT-SCD), KU Leuven, Leuven, 2013.

[10] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue. TIMIT Acoustic-Phonetic Continuous Speech Corpus, 1993.

[11] P. Golik, P. Doetsch, and H. Ney. Cross-entropy vs. Squared error training: A theoretical and experimental comparison. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2(2):1756–1760, 2013.

[12] A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks, Phd Thesis*. 2008.

[13] A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin Heidelberg, 2012.

[14] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, pages 1–43, 2013.

[15] A. Graves, S. Fernandez, F. Gomez, and J. Schmidhuber. Connectionist Temporal Classification : Labelling Unsegmented Sequence Data with Recurrent Neural Networks. *Proceedings of the 23rd international conference on Machine Learning*, pages 369–376, 2006.

[16] A. Graves, A.-R. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, (6):6645–6649, 2013.

[17] A. Graves, G. Wayne, and I. Danihelka. Neural Turing Machines. *Arxiv*, pages 1–26, 2014.

[18] S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, 1998.

[19] S. Hochreiter and J. Schmidhuber. LONG SHORT TERM MEMORY. *Technical Report FKI-207-95*, pages 1–8, 1995.

[20] X. Huang, A. Acero, and H.-W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm and System Development*. Prentice Hall PTR, Redmond, 2001.

[21] B. H. Juang, L. R. Rabiner, and J. G. Wilpon. On the use of bandpass filtering in speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(7), 1987.

[22] B. Langmead. Dynamic programming and edit distance, 2016.

[23] K. F. Lee and H. W. Hon. Speaker-independent phone recognition using hidden Markov models. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(11):1641–1648, 1989.

[24] A. Neelakantan. NEURAL PROGRAMMER: INDUCING LATENT PROGRAMS WITH GRADIENT DESCENT. Number 2005, pages 1–18, Puerto Rico, 2016. ICLR 2016.

[25] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *Proceedings of The 30th International Conference on Machine Learning*, (2):1310–1318, 2012.

[26] R. Rojas. *Neural Networks - A Systematic Introduction - Backpropagation.* Springer Berlin Heidelberg, 1996.

[27] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Over tting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[28] G. van Rossum, B. Warsaw, and N. Coghlan. PEP 8 – Style Guide for Python Code, 2001.

[29] S. Wager, S. Wang, P. Liang, and C. Science. Dropout Training as Adaptive Regularization. *Advances in Neural Information Processing Systems*, 26:351—-359, 2013.

[30] P. Woodland, G. Evermann, and M. Gales. HTKBook, 2006.

Master thesis filing card

Student: Moritz Wolter

Title: Building an end-to-end speech recognizer

UDC: 621.3

Abstract:

In the past machine learning relied heavily on algorithms designed by experts to solve a specific task. Which lead to highly sophisticated algorithms, which could be grasped only by small groups of people. The human brain however does not work this way, although specialized areas exist, these areas consist of similar building blocks. Artificial neural networks attempt to mimic this layout. Similar algorithmic structures are used for a wide variety of tasks. This thesis deals with the application of neural networks in speech recognition. Replacing the various subsystems by one integrated network based approach.

Thesis submitted for the degree of Master of Science in Mathematical Engineering

Thesis supervisors: Prof. dr. ir. Johan Suykens
Prof. dr. ir. Hugo Van hamme

Assessors: Prof. dr. ir. Raf Vandenbrid
Prof. dr. ir. Patrik Wambacq

Mentor: Ir. Vincent Renkens