Karlsruhe Institute of Technology

Revised Speech Chain Loop featuring Quality Estimation

Bachelor thesis of

Oliver Wirth

At the Department of Informatics
Institute of Anthropomatics and Robotics,
Karlsruhe Institute of Technology

In cooperation with
Augmented Human Communications Laboratory,
Nara Insitute of Science and Technology

Reviewer:	Prof. Dr. Alexander Waibel
Second reviewer:	Prof. Dr.-Ing. Tamim Asfour
Advisor:	Dr. Sebastian Stüker

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

I followed the rules for securing a good scientific pracise of the Karlsruhe Institute of Technology (Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT)).

Karlsruhe, 11.05.2020

. . . . . .

(**Oliver Wirth**)

Abstract

In 2017, Tjandra et al. [37] introduced a novel approach to the simultaneous training of an automatic speech recognition (ASR) system and a text-to-speech (TTS) system, called the Speech Chain. This system incorporates the same speech perception and production behaviors as humans, by training the ASR and TTS systems in a closed loop. The Speech Chain thereby enables the components to train on unlabeled data, which significantly improves the performance in comparison to systems, that are solely trained on labeled data. However, this approach relies on sufficiently good predictions of the individual, pre-trained components, because faulty hypotheses of the ASR can influence the TTS learning process in an undesired way, and vice versa.

In this Bachelor thesis I present two approaches that aim to improve the performance of the Speech Chain. The first enhancement introduces an external quality estimation (QE) component into the Speech Chain, with the goal of estimating the quality of ASR hypotheses. With the second enhancement, I tighten the connection between ASR and TTS, by sharing their respective input/output layers during the unsupervised training step. The proposed modification allows the Speech Chain to filter bad transcriptions produced by the ASR, that would guarantee a high loss for the TTS, even though its generated speech might be comparatively good.

The experimental results show, that while the quality estimator is able to properly distinguish good from bad quality, the Speech Chain as a whole does not necessarily benefit from that.

Contents

Abstract v

1 Introduction 1

2 Background 3
2.1 Speech Recognition/Synthesis . . . 3
2.1.1 Speech Recognition . . . 3
2.1.2 Speech Synthesis . . . 4
2.2 Machine Learning . . . 5
2.2.1 Classification . . . 5
2.2.2 Artificial Neural Network . . . 8
2.2.3 Multilayer Perceptron . . . 10
2.2.4 Convolutional Neural Network . . . 11
2.2.5 Recurrent Neural Network . . . 12
2.2.6 Long Short-Term Memory . . . 13
2.2.7 Sequence-To-Sequence Model . . . 15
2.2.8 Hyperparameters . . . 18

3 Related Work 21
3.1 Speech Chain . . . 21
3.1.1 Training Process . . . 21
3.1.2 ASR Model . . . 22
3.1.3 TTS Model . . . 22
3.1.4 Experimental results . . . 23
3.1.5 End-to-End Loss via Straight-Through Estimator . . . 24
3.2 Quality Estimation . . . 25
3.2.1 Automatic Speech Recognition . . . 25

4 Method 27
4.1 ASR Quality Estimation . . . 27
4.1.1 Classification vs. Regression . . . 28
4.1.2 Baseline . . . 28
4.1.3 Manual Feature Extraction . . . 29
4.1.4 MLP Model . . . 30
4.1.5 Recurrent Model . . . 30
4.1.6 Multimodal Model . . . 31
4.2 Input/Output Layer Sharing . . . 31
4.2.1 ASR to TTS Iteration . . . 32
4.2.2 TTS Modification . . . 32
4.2.3 TTS to ASR Iteration . . . 33

5 Experiments 35
5.1 Dataset . . . 35

5.2 Hyperparameters 36
5.2.1 QE Hyperparameters 36
5.2.2 Speech Chain Hyperparameters 36
5.3 Results 37
5.3.1 QE Results 37
5.3.2 Speech Chain Results 38

6 Conclusion **41**
6.1 Further Work 42

Bibliography **43**

Glossary **47**

Chapter 1

Introduction

The Machine Speech Chain [37] successfully models the auditory feedback loop described by Denes et al. [10]. Speech output produced by a TTS is recognized by an ASR, just like a human hearing his own speech. Also, text transcriptions recognized by an ASR are transformed into speech, similiar to humans trying to repeat a sentence somebody else has said. Nevertheless, the speech chain process in the human brain is vastly more complex, since the brain has a wide variety of tasks and capabilities. One of the things the current Speech Chain implementation cannot do, is the assessment of the quality of its recognized speech. Humans on the other hand, are able to judge whether they properly understood an utterance or not. This is possible by taking various factors into account, that can influence the speech recognition ability, including background noise, the speaker's accent, visibility of the speaker, and others.

Although this kind of information is not available in the scenario of the Speech Chain framework, this work aims to design and implement a quality estimator, that decides whether a given ASR transcription is good enough for training. This QE component is implemented by using machine learning techniques, namely neural networks. Secondly, to further tighten the loop connection between ASR and TTS, an input/output layer sharing mechanism is proposed. In combination with the QE, this can potentially improve the learning process of the ASR, as the TTS error can be backpropagated through the ASR as well.

Chapter 2

Background

This chapter provides a short overview over technology and concepts used in this thesis. First, I will present the basics of automatic speech recognition and synthesis. Next, I will explain the concepts of machine learning and neural networks, that are required in order to comprehend the inner workings of the Speech Chain.

2.1 Speech Recognition/Synthesis

Both speech recognition and speech synthesis are core fields in computational linguistics. Speech recognition describes the task to transcribe spoken words into a textual representation, while speech synthesis can be seen as an inverse task, to produce speech from a given transcription. These disciplines are commonly used in a variety of everyday applications, such as personal assistants, navigation systems, chatbots, handicapped accessible systems, dictation, and others.

This section is based on Huang et al. [17].

2.1.1 Speech Recognition

Automatic speech recognition is a difficult task for machines. While humans are naturally well equipped for this kind of task, machines struggle with the variability of human speech. Therefore, it is possible for the same word to produce vastly distinct waveforms, dependent on the speaker, the speaker's accent and prosody, and the speaking environment.

Classic ASR systems feature signal processing, an acoustic model and a language model. Signal processing usually involves taking the Fourier transform of short time frames and transforming them to a meaningful representation, like the Mel scale. The acoustic model is a representation of the relationship between an audio signal and linguistic units, like phonemes. It is used to model the probability $P(X|W)$ of the audio signal X, given it represents the word sequence W. The language model assigns a probability $P(W)$ to a sequence of words W. A common approach is to assume the probability of a word to only depend on the previous n words, this is known as an n-gram model.

Finding a transcription for a given speech sample can be seen as a statistical classification problem: The transcription is the class $\hat{W}$ the ASR searches for, given the acoustic observation X. With Bayesian statistics, the ASR problem can be formulated as:

$$\begin{aligned}\hat{W} = \underset{W}{\operatorname{argmax}} P(W|X) &= \underset{W}{\operatorname{argmax}} \frac{P(X|W) * P(W)}{P(X)} \\ &= \underset{W}{\operatorname{argmax}} P(X|W) * P(W)\end{aligned}$$

The class-based probability $P(X|W)$ is modeled by the acoustic model and the prior probability $P(W)$ by the language model. To avoid calculating the argmax function over all possible classes W, a feasible decoding algorithm is needed.

To efficiently predict $P(X|W)$, HMM-GMM models are used. Hidden Markov models (HMMs) allow modeling the transition between phones, by treating speech as a Markov chain. With decoding algorithms, like the Viterbi algorithm, it is possible to efficiently calculate the argmax function. To train such HMM-GMM models, a special variant of the expectation maximization algorithm can be used. The Baum-Welch algorithm is able to iteratively find a near optimal parameter set.

With the recent boom of deep learning and neural networks, ASR underwent a metamorphosis as well. Hybrid systems were developed, combining the advantages of neural nets and HMMs to create HMM-ANN models [3]. Recent research even tends to implement ASR systems that don't need an HMM at all. These sequence-to-sequence models 2.2.7 are pure neural networks, that, while still in need of signal preprocessing, directly model the posterior probability $P(W|X)$ without the need of separate acoustic and language models [6].

2.1.2 Speech Synthesis

A TTS' task to convert a textual representation into speech seems comparatively easy at first glance. However, producing the speech form of independent phonemes is one thing, finding a coherent prosody to produce naturally sounding words and phrases is another. Huang et al. [17] compare this to trying to drop a foreign accent when speaking a non-native language.

Classic TTS systems also consist of several components. A text analysis component is responsible for normalizing text, by removing punctuation and replacing numbers, etc. with their written counterparts. Next, a phonetic analysis is performed on the normalized text, which turns words into phoneme sequences. This is followed by a prosodic analysis component, attaching pitch, duration and other variability information to the sequence. In the final step, the actual speech synthesis unit converts the given sequence with all its parameters into a speech waveform.

As for the ASR, TTS research likewise experienced a surge in new approaches and technologies, by the recent advent of deep learning. Current state-of-the-art TTS systems, like Tacotron by Wang et al. [39], use sequence-to-sequence models, that eliminate the need of separate components with individual tasks, by directly predicting the speech waveform.

A common metric to evaluate the performance of an ASR system, is the word error rate (WER) [24]. It is calculated as a fraction of the word-based Levenshtein distance and the length of the reference.

$$WER(W_{hyp}, W_{ref}) = \frac{dist_{edit}(W_{hyp}, W_{ref})}{|W_{ref}|}$$

The Levenshtein distance $dist_{edit}(x, y)$, also known as edit distance, is the minimum number of edits required to transform the sequence x into the sequence y. Edits in this context are single insertions, deletions or substitutions. It can be calculated by dynamic programming and usually operates on character level, but can be used on any type of token, like words in the case of WER. If the WER is desired on a character level, it can be calculated analogically, and is called character error rate (CER).

2.2 Machine Learning

This section provides some insights into most of the technology and concepts present in this thesis. The core theme is machine learning, which describes algorithms and methods for the artificial creation of knowledge, usually by adapting a function approximator to some kind of dataset. This adaption process is called training. Naturally, ASR and TTS systems, as introduced in the previous section 2.1, also belong to the broad category of machine learning, as their parameters are also optimized in a training process.

Pattern recognition is a field that has a deep bond to machine learning, as demonstrated with the prominent example of handwritten digit recognition. This task is difficult to solve with more classic algorithms, that involve rule-based image processing or heuristics for the location and shapes of strokes. The reason for this difficulty lies in the variability of handwriting, which makes it difficult to cover all edge cases by such an algorithm. By applying machine learning methods like support vector machines (SVMs) or neural networks however, the performance of such a recognizer can be greatly improved, without the need to handcraft rules and heuristics. This can be seen by looking at the experiments on the popular MNIST dataset, that contains around 60,000 images [23] for training. In their original paper, Lecun et al. [23] propose a SVM that achieves an error rate of only 0.8%, while neural network approaches manage to top that score by achieving an error rate of under 0.3% [7].

Machine learning can be divded into supervised and unsupervised methods. In supervised training, every data sample x has a ground-truth label t. For a function approximator f, this allows for the precise calculation of the error between the prediction $f(x)$ and the label t, which in turn makes it easy to formulate an intuitive training objective. SVMs, Perceptrons and most neural network approaches require supervised training. However, these labels are often annotated by hand, which makes that kind of labeled data expensive. Unsupervised systems do not have those ground-truth labels, and are required to "organize themselves" in a meaningful way. This includes clustering algorithms, like the k-nearest neighbor algorithm.

This section is based on Bishop [2], as well as Goodfellow et al. [12].

2.2.1 Classification

When recognizing patterns, it is often useful to assign them to a finite number of classes. This allows for generalization and grouping of patterns. The process of finding the right class for a given pattern is called classification. A prime example would be the handwritten digit recognition 2.2, where the ten classes are the ten different digits to be recognized.

In some cases it is desirable for a function approximator to not output a discrete class, but rather a continous value. This task is called regression, and is used for predicting continous functions.

Classifiers can be divided into two sub-categories. Linear classifiers can only find correct class labels if the dataset is linearly separable, meaning that the input space can be partitioned by linear decision boundaries, so that each partition only contains datapoints of

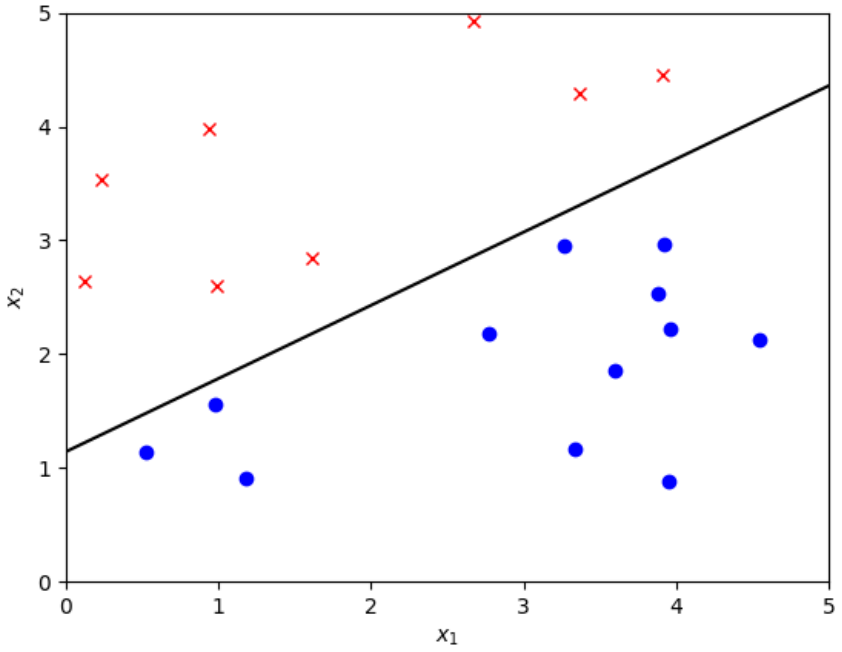

Figure 2.1: Data points belonging to two different classes, separable by an exemplary decision boundary (black)

a single class. For a d-dimensional input space, these decision boundaries have the shape of $(d-1)$-dimensional hyperplanes. A two-dimensional example can be seen in figure 2.1.

However, linear classification models are often not sufficient for real-world applications. The XOR function is deceptively easy to calculate for humans and machines alike, but its space is not linearly separable, as can be seen in figure 2.2.

$$XOR(x, y) := x \oplus y = \begin{cases} 0, & x = y \\ 1, & x \neq y \end{cases}$$
$$x, y \in \{0, 1\}$$

Classifiers that are able to draw higher-order decision boundaries are called non-linear. More advanced neural networks are non-linear classifiers, whereas single Perceptrons and SVMs are linear classifiers.

Low-dimensional examples, like in figure 2.1, have the advantage of being intuitive, so that a new datapoint could be easily assigned to a new class by just looking at the rest of the data. But this intuitivety quickly disappears as the number of dimensions grows. Not only does a high-dimensional input space cause problems for humans to imagine, it also poses a problem for classifiers, as the significance of single dimensions with respect to the classes is usually different. That means that datapoints, which have a comparatively small euclidean distance, can still belong to different classes, just because they have a high distance in a single, but important dimension. This problem, which arises in high-dimensional datasets, is called the curse of dimensionality [1].

There are techniques, such as the principle component analysis (PCA) [16], which mitigate the curse of dimensionality by reducing the dimensionality of the input space. PCA

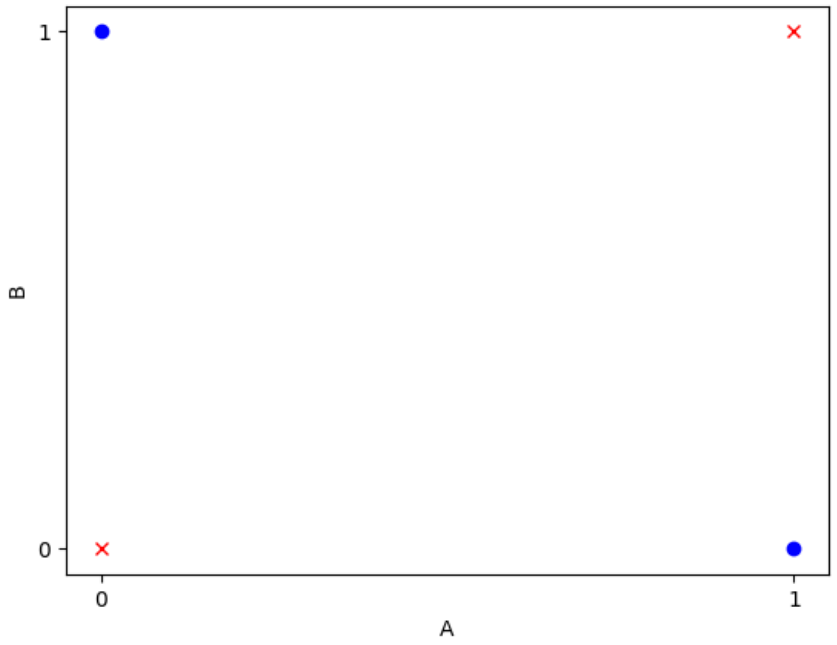

Figure 2.2: Function values of the binary XOR function $A \oplus B$; there is no linear function that can perfectly separate the classes of $A \oplus B = 0$ (red crosses) and $A \oplus B = 1$ (blue dots)

is the orthogonal projection of the data onto a subspace with a lower dimensionality. The projection is chosen, so the variance of the projected data is maximized, and therefore as less information as possible is lost in the process. Another method is Fisher's linear discriminant analysis, that also reduces dimensionality by projecting the data onto a subspace, in order to maximize the distance of classes in the projection. Since the classes must be known for this approach, Fisher's linear discriminant analysis is only applicable in a supervised setting.

Classification tasks, that only involve two classes, are called binary classification tasks. As a simplification, these classes are called "positives" and "negatives" in the following. For the evaluation of such binary classifiers, four different cases can be distinguished:

- True positives: the classifier predicted positive, which was also the ground-truth
- False positives: the classifier predicted positive, but it was actually negative
- False negatives: the classifier predicted negative, but it was actually positive
- True negatives: the classifier predicted negative, which was also the ground-truth

Based on these cases, several metrics can be calculated to better express their relations. The precision of a classifier is the ratio between correct positive classifications and all positive classifications [5] [40]. It answers the question of how many of the samples selected by the classifier are positive.

$$precision = \frac{\#[\text{true positives}]}{\#[\text{true positives}] + \#[\text{false positives}]}$$

The ratio between true positives and all positives samples, correctly predicted and not, is called recall [5] [40]. This metric measures how many of the positive samples are selected by the classifier.

$$recall = \frac{\#[\text{true positives}]}{\#[\text{true positives}] + \#[\text{false negatives}]}$$

Although they are both intuitive metrics, they are not always relevant to the actual problem. The precision can be 1 or close to 1, if the classifier acts very restrictive and only selects a small number of samples. That does not guarantee a "good" classifier however, as it possibly throws away a large portion of perfectly good samples. In a setting where it is more important to not select negative samples, precision would be more relevant than recall. On the other hand, a high recall could also mean that the classifier acts overly lenient and, in an extreme case, simply classifies every sample as positive.

The F_1 score provides a good compromise between precision and recall [5] [40]. It is the harmonic mean of those two metrics.

$$F_1 = \frac{2}{precision^{-1} + recall^{-1}} = 2 * \frac{precision * recall}{precision + recall}$$

Another prominent metric is accuracy. A classifier's accuracy is the percentage of all correctly classified samples, positive and negative.

$$accuracy = \frac{\#[\text{true positives}] + \#[\text{true negatives}]}{\#[\text{all samples}]}$$

2.2.2 Artificial Neural Network

To explain what neural networks are made of, it is necesseary to look at the Perceptron algorithm. First introduced by Rosenblatt [29], the Perceptron is a mathematical model of a neuron. Much like a neuron in biology, a Perceptron fires, if it receives a stimulus exceeding a certain threshold. Speaking in mathematical terms, the input synapses are represented by a weight vector w, and the trigger threshold by a scalar value b'. For simplicity reason, a bias value $b := -b'$ is often used instead of the threshold. Together with an activation function that forces the Perceptron to output a binary value (0 or 1), also known as the Heaviside or step function Θ, the Perceptron function f can be described as follows.

$$f(x) = \Theta(w \cdot x + b) = \begin{cases} 0, & w \cdot x < b' \\ 1, & w \cdot x \geq b' \end{cases}$$
$$x, w \in \mathbb{R}^n,\ b \in \mathbb{R}$$

The Perceptron realizes a binary, linear classifier. That makes it impossible to learn the XOR function 2.2, or other non-linear classification problems with this algorithm, but it is applicable for linearly separable problems.

The Perceptron is able to learn from a training dataset. In the following paragraphs, x', w' are defined as $x' := \begin{pmatrix} 1 \\ x \end{pmatrix} \in \mathbb{R}^{n+1}$ and $w' := \begin{pmatrix} b \\ w \end{pmatrix} \in \mathbb{R}^{n+1}$, so that $w \cdot x + b = w' \cdot x'$.

By naming the classes ω_0 and ω_1, so that x is classified as $\omega_{f(x)}$, all samples x' from the dataset $X' \subseteq \mathbb{R}^{n+1}$, that belong to class ω_0, can be set to $-x'$. Now, all vectors are classified correctly if $\forall x' \in X' : w' \cdot x' \geq 0$. The weights w' can be iteratively updated until the set of misclassified samples M is empty:

- for all $x' \in X'$:
- if $w' \cdot x' < 0$, then set w' to $w' + \eta * x'$
- repeat if w' changed

$\eta \in \mathbb{R}$ is called the learning rate. Given the Perceptron convergence theorem [2], this algorithm converges if there is an exact solution, meaning X is linearly separable.

An alternate solution of the classification problem would be to formulate an objective function L by measuring the error, or loss, of the Perceptron. Finding optimal weights w' is then equal to the minimization of L. There are various approaches to minimizing functions, one of the most broadly applicable ones is gradient descent. Its only requirement is the differentiability of the objective function. For the Perceptron, the least-squared distance, also called mean squared error (MSE), can be used as a loss function L.

$$L(w') = \frac{1}{2} \sum_{x \in X} (t_x - f(x))$$

In this case, t_x is the ground-truth label of the sample x. Other widely used loss functions in neural networks include the cross-entropy function, which is useful in multi-class problems where the network output is comprised of the class probabilities.

Based on this loss, the gradient descent algorithm can now calculate the derivative of L with respect to w', and move the weights in the direction to the negative gradient $-\nabla L$. This is an intuitive step, as the gradient of a function always points into the direction of the steepest ascent. It can be done by exploiting the chain rule the following way:

$$\frac{\partial L}{\partial w'} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial w'}$$
$$\text{neuron activation } a := f(x)$$

However, to calculate $\frac{\partial a}{\partial w'}$, another problem emerges, as the Heaviside function Θ is not differentiable. As a solution, a smooth approximation of Θ is used, the sigmoid function σ.

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

Like the loss function, the choice of a good activation function is application specific. Prominent examples are the tangens hyperbolicus, that smoothly approximates the sign function, or the rectified linear unit (ReLU), that behaves like a linear function for $x > 0$, and has seen lots of application in recent convolutional neural networks. Another important function is the softmax function, as it transforms a multidimensional vector into

a probability vector of the same dimension, which can be used for multi-class classification tasks.

With differentiable loss and activation functions, it is finally possibly to train the Perceptron by using gradient descent. The advantage of this approach is its universality, as it allows calculating weight gradients of not only single neurons, but arbitrary neuron architectures. For example, organizing these Perceptrons into multiple layers results in a multilayer Perceptron, which is a first actual neural network. The process of calculating the loss gradient with respect to the weights of multiple layers is called backpropagation, and is a crucial algorithm for the efficient training of neural networks.

2.2.3 Multilayer Perceptron

As already mentioned, a multilayer Perceptron (MLP) is built by coupling several Perceptron neurons. They are usually organized into layers, which are then stacked behind each other. A MLP with one hidden and one output layer is displayed in figure 2.3. This type of network is also known as a feed-forward network, coming from the forward-only direction of the dataflow.

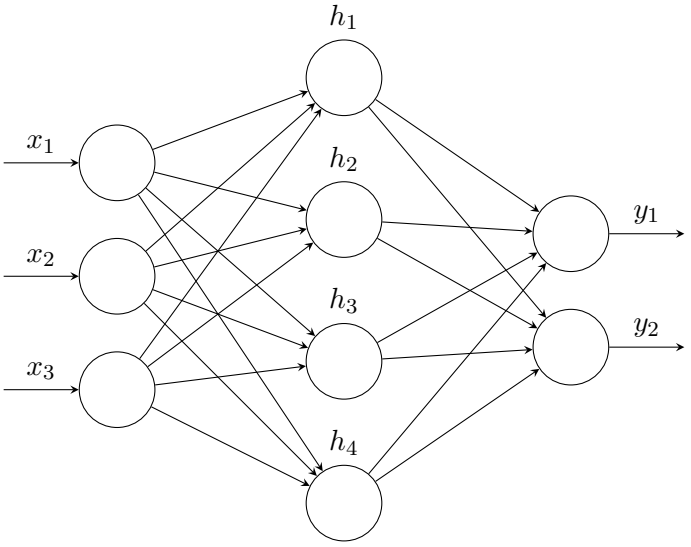

Figure 2.3: Simple feed-forward network architecture with one hidden layer

In the example, the network accepts a three-dimensional input vector x, and after passing both hidden and output layer, it outputs a two-dimensional vector y. Both hidden and output layer feature weight matrizes $W^h \in \mathbb{R}^{3\times4}$, $W^o \in \mathbb{R}^{4\times2}$ and biases $b^h \in \mathbb{R}^4$, $b^o \in \mathbb{R}^2$, as well as an activation function, e.g. the sigmoid, which is applied pointwise.

$$
\begin{aligned}
f(x) &= y \\
y &= \sigma(W^o \cdot h + b^o) \\
h &= \sigma(W^h \cdot x + b^h)
\end{aligned}
$$

While the Perceptron is only a linear classifier, a MLP is a non-linear classifier, due to multiple layers with non-linear activation functions inbetween. This realization has revived the interest in the neural model [2], almost 30 years after the Perceptron was first

proposed. Neural networks are so called universal approximators, meaning they can in theory approximate any continous function on a compact input space, given they have at least one hidden layer and a sufficient number of hidden units. In practice, although one hidden layer is hypothetically speaking enough, it has been shown that multiple smaller hidden layers improve the convergence time and performance significantly [2].

2.2.4 Convolutional Neural Network

Classic image processing tasks involve the use of so called filters, usually represented by a weight matrix. Applying a filter to an image equals using that matrix as a stencil in each pixel location, and multiplying pixel values with the matrix weights, before summing them up. This process is also known as a (discrete) convolution. These filters allow for the detection and highlighting of edges and corners in images, as well as simply adjusting brightness and contrast. With the advance of neural networks and deep learning, many computer vision tasks, like image classification, moved in this direction as well. One of the most influential inventions is the convolutional neural network. Compared to regular feed-forward networks, it features so called convolutional layers, that apply a filter matrix in the same way as described earlier. The difference is, that the weights of such a filter matrix are treated as trainable parameters. That way, the network can learn itself which kind of features it would like to extract from the input, instead of handcrafting these filters.

Convolutional neural networks (CNNs) have the advantage of considering the spatial features of its input, what makes them invariant to certain transformations, like scaling, translations and rotations in some cases. Consider the already mentioned handwritten digit recognition 2.2. The digit 6 for example, should still be recognized during inference, even if its training samples had a different scaling and position. The classifier should also be robust towards small tilts and rotations, as long as they don't exceed a certain angle. This invariance comes from the fact, that the convolutional layers extract local features, regardless of their position, as the filter shares its weights for each location it is applied.

A convolutional layer consist of k filters, also called kernels, each with the size $w \times h$ and a scalar bias weight. Each kernel is applied to a location and then moved s pixels to its next location. The stride s is a hyperparameter like k, w, h. To avoid shrinking the image through subsequent convolutional layers, an edge padding p can be used for the kernel locations. The image resulting from the convolutions of one of the kernels is then stacked with the results of the other kernels and fed into the next layer. In practice, convolutional layers are often used alternating with pooling layers. Pooling layers serve the purpose of decreasing the image size and thereby reducing the number of activations and necessary parameters. They can be implemented by taking the maximum, minimum or average of small $s \times s$ patches of the image, the same as the stride s, which would result in an image that is $\frac{1}{s}$ of the previous size.

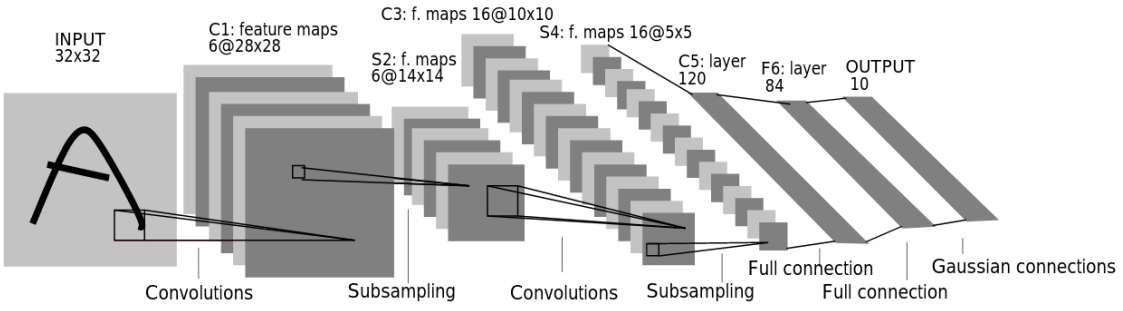

Figure 2.4: LeNet-5 for digit recognition [23]

For regression or classification tasks, the final layers of a CNN usually consist of one or multiple fully-connected layers. In the scenario of handwritten digit recognition, this would mean at least one hidden layer and a ten-dimensional output layer. Lecun et al. [23] achieve with their initially proposed CNN architecture, shown in figure 2.4, an error rate of 0.7, which has been topped multiple times by other CNNs since their publication. A CNN can be trained as a whole by using the backpropagation algorithm, while considering the constraint of shared weights of the spatial kernel applications.

2.2.5 Recurrent Neural Network

What classic feed-forward neural networks lack, is the ability to preserve their context. A MLP or CNN will instantly "forget" its previous activations when given a new input. This is not necessarily an unwanted behaviour, but it impedes tasks where this context matters. For a biological example, take the human brain: If we would forget all our thoughts immediately, we could not listen to a conversation or watch a movie, as we could only process words and images one at a time, without knowing the context of these words or images. As this example illustrated, speech processing, amongst other fields, needs this kind of context building mechanism. For that means, recurrent neural networks pose a solution.

A recurrent neural network (RNN) has recurrent connections between its neurons and layers, which go against the usual dataflow in feed-forward networks. At a given timestep, these connections carry the activations of the previous timestep, which provides the network information about the preceding inputs. This allows a RNN to process sequences, as single units of the sequence, e.g. words, characters or speech frames, are given to the network one at a time, but the network has an understanding of the context around the current input and can act accordingly. How the recurrent connections are implemented depends on the type of network. Two of the most simple architectures are Elman and Jordan networks. An Elman network features an architecture similiar to the network in figure 2.3, but it has a loop connection around the hidden layer, meaning the hidden layer receives, additionally to the input from the current timestep, its own activations from the previous timestep as an input. Jordan networks also have a similiar architecture, but they have a recurrent connection from the output layer to the hidden layer. Both networks are displayed in figure 2.5.

Training a network with recurrent connections requires a slight alteration of the backpropagation algorithm. The network is unrolled over all timesteps, as shown for an Elman network in figure 2.6, and each node in the unrolled network graph is treated independently. Parameters are shared over each timestep, so the total weight update is the sum of the updates through all timesteps. This is called backpropagation through time.

However, during backpropagation through a recurrent network architecture, individual gradients can get quite small, as many widely-used activation functions have gradients close to zero, like $\frac{d}{dx}\tanh(x) \in (0, 1]$ or $\frac{d}{dx}\sigma(x) \in (0, 0.25]$. Such activation functions also saturate quite fast, as a potentionally large difference in input can amount to only a minor change in activation output, e.g. $\sigma(10) \approx 0.99995 \approx 0.99999 \approx \sigma(100)$. This behaviour, paired with long product chains that arise through repeated use of the chain rule, forces gradients in later timesteps to become too small to cause a significant change in the network parameters. Therefore, the learning process of the network can stagnate or stop completely, which is known as the vanishing gradient problem. It is not limited to recurrent networks, as it can also occur in very deep feed-forward architectures. Similiarly, the exploding gradient problem occurs in long backpropagation chains with many gradients greater than 1, causing the parameters to not converge because of the too large weight updates.

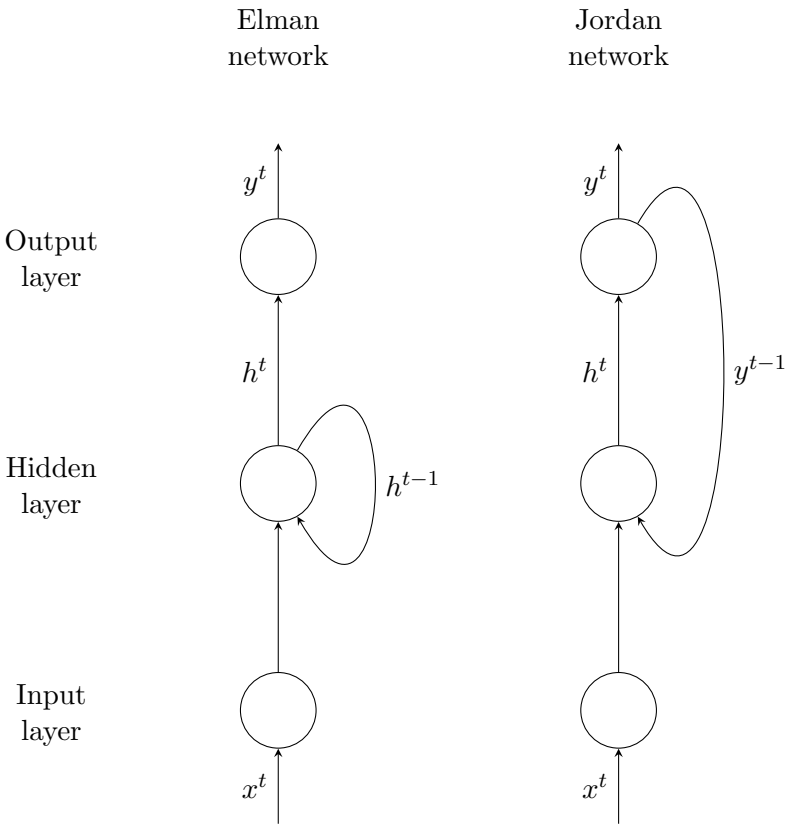

Figure 2.5: Basic RNNs compared: Elman network on the left, Jordan network on the right

There are several approaches to mitigate these problems. A first solution could be the exchange of activation functions for functions with better gradient properties, like the one-sidedly non-saturating ReLU. Also worth mentioning is the introduction of normalization layers, that normalize intermediate gradients, thereby preventing them from becoming too small or too large. In case of the exploding gradient problem, it is possible to introduce L1/L2 regularization terms into the loss function, which penalizes large weights, or implement gradient clipping, to clip gradients as soon as their norm exceeds a certain threshold.

More recently, He et al. [14] proposed residual connections, that can be visualized as forward edges in the network graph, skipping one or more layers before adding a previous activation on the last layer output. During gradient descent, those connections prevent the gradient from getting too small, through the addition of an earlier gradient term, which helps in cases of vanishing gradients. Specifically for RNNs, a prominent solution of the vanishing gradient problem, is the long short-term memory (LSTM), which will be presented in the next section 2.2.6.

2.2.6 Long Short-Term Memory

The previously discussed RNNs 2.2.5, are able to process sequences with arbitrary length, at least in theory. But the vanishing and exploding gradient problems prevent too long-range dependencies from being learned. LSTMs [15] solve this problem by introducing a memory cell and three gates. The memory cell carries information from one timestep to the next, and can be changed by the input and forget gates. New information is saved in the memory cell by the input gate, if the LSTM unit decides that the input is worth remembering. The forget gate decided when to reset certain information in the memory

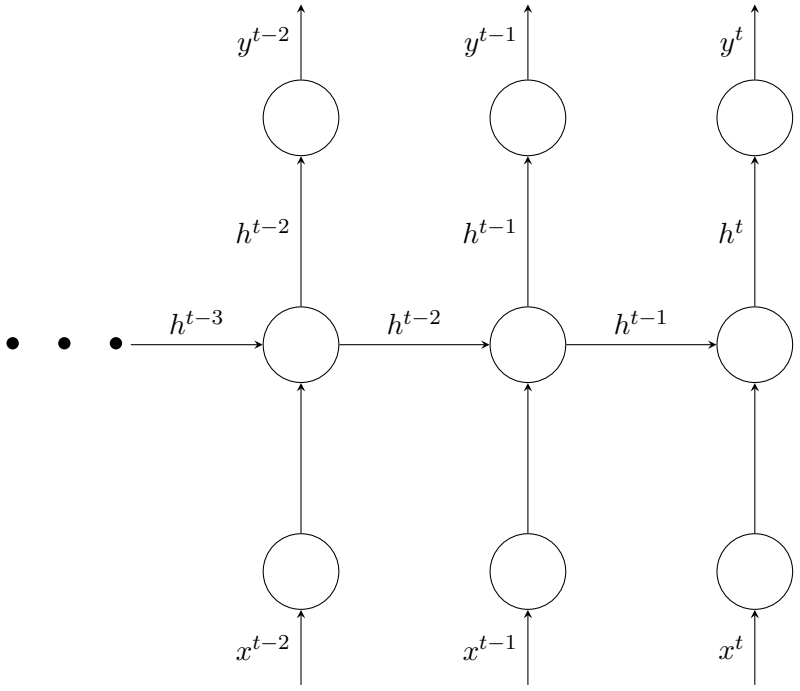

Figure 2.6: Unrolled Elman network at timestep t

cell, as it may no longer be needed. Lastly, an output gate is responsible for the output of the LSTM at the current timestep.

The memory cell of a LSTM unit has a self connection, like the vanilla RNNs. By using this kind of dedicated memory cell with its respective gates, a LSTM assures a constant error flow during backpropagation, which hinders gradients from vanishing. However, it may still suffer from the exploding gradient problem. For that reason, many LSTM implementations also implement gradient clipping, to get rid of exploding gradients.

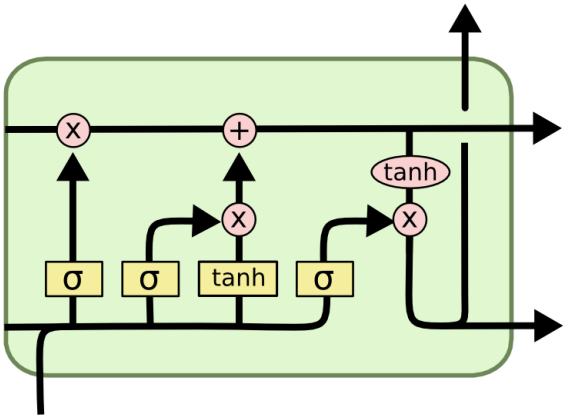

Figure 2.7: LSTM architecture, the upper horizontal line represents the memory c, and the yellow rectangles represent the gates. From left to right: forget gate f (σ), input gate i (σ), candidate layer $\tilde{c}$ (tanh), output gate o (σ) [27]

The gates consist of fully-connected layers with an activation function, usually the sigmoid or tangens hyperbolicus. Layer weights, and therefore the way gates operate, are trained together with the network. Gate activations in a forward pass are calculated as follows:

$$x' := \begin{pmatrix} h^{t-1} \\ x^t \end{pmatrix}$$
$$f^t = \sigma(W^f \cdot x' + b^f)$$
$$i^t = \sigma(W^i \cdot x' + b^i)$$
$$o^t = \sigma(W^o \cdot x' + b^o)$$

The forget gate f^t works multiplicative: When it is 0, memory is reset, in case of 1, it is kept. As for the input gate i^t, candidate values $\tilde{c}^t$ created by a tanh layer, before eventually adding them onto the memory cell c^{t-1}.

$$\tilde{c}^t = \tanh(W^{\tilde{c}} \cdot x' + b^{\tilde{c}})$$
$$c^t = f^t \odot c^{t-1} + f^t \odot \tilde{c}^t$$

The LSTM unit outputs its hidden state h^t at timestep t, controlled by the output gate o^t.

$$h^t = o^t \odot \tanh(c^t)$$

Sigmoid and tangens hyperbolicus functions in the equations above are applied pointwise, and the operator $\odot$ denotes the Hadamard product. A schematic drawing of the LSTM's architecture is depicted in figure 2.7.

As already mentioned, LSTMs, or RNNs in general, are widely used in neural models for speech processing. However, natural language sentences often contain dependecies in both directions, not only backward dependencies. Such forward dependecies are not representable by a standard LSTM. This can be resolved by applying a trick: A second LSTM with separate parameters is fed the same sequence as input, but reversed [13]. Their resulting output is then stacked, before it can be processed by the next layer. The two LSTMs are called forward and backward LSTM, and together they form a bidirectional LSTM, or Bi-LSTM, which is able to capture dependencies in both directions. This concept is not unique to LSTMs, but can be applied to vanilla RNNs as well [30].

LSTMs have a wide variety of applications, including robot control, handwriting recognition and ASR. In addition to solving the vanishing gradient problem, LSTM-based network architectures also outperform vanilla RNNs in many areas. Shewalkar et al. [31] compared ASR systems employing neural networks in the form of vanilla RNNs and LSTMs against each other. In their experiments on a 500 neuron hidden layer model, LSTMs turned out to have a 10% lower WER than their vanilla counterpart. LSTMs are also an integral part of the more advanced sequence-to-sequence models, like the Transformer model proposed by Vaswani et al. [38].

2.2.7 Sequence-To-Sequence Model

Processing sequences is a great ability of RNNs, as described earlier 2.2.5, and is used widely for sequence classification or time series prediction. Nevertheless, RNN models introduced until now always produce exactly one output per timestep. Therefore, a sequence generated by an ASR has the same length as the input sequence. But for applications like

speech recognition, machine translation, chatbots and many more, a variable length sequence output is necessary.

A solution for such sequence-to-sequence problems was presented by Sutskever et al. [34]. Their sequence-to-sequence model consists of an encoder and decoder component. The encoder is a RNN, that processes an input sequence X and maps it to a fixed-size vector v, representing the input. In general, any RNN is a feasible choice, however, due to the aforementioned advantages of LSTMs 2.2.6, the authors used a multilayered LSTM encoder. The representation vector is acquired by taking the LSTM hidden state of the last timestep M.

$$v = encode(X_1, \ldots, X_M)$$

The decoder is also a RNN, specifically, in the work of Sutskever et al. [34], a multilayered LSTM, that outputs a sequence based on the input representation. To generate that output sequence Y, the decoder receives a designated starting token Y_{start}, in language applications usually a beginning-of-sentence token, and generates the first token Y_1 of the sequence. In the next timestep, Y_1 is used as the decoder input to generate Y_2, and so forth.

$$\begin{aligned} Y_1 &= decode(Y_{start}) \\ Y_n &= decode(Y_{n-1}) \\ n &\in \mathbb{N} \end{aligned}$$

This process is repeated until the decoder generates a special ending token Y_{end}, to mark the end of the sequence.

The encoder-decoder architecture can be trained with the same backpropagation through time approach used for RNNs. The loss is calculated on a per-token basis, by comparing the generated token Y_n at timestep n to the ground-truth token Y_n^{ref}. A popular method for the training of sequence-to-sequence models is teacher-forcing [12]. During training time, the decoder thereby receives the ground-truth tokens as input, instead of the previously generated output token.

$$\begin{aligned} Y_n &= decode(Y_{n-1}^{ref}) \\ n &\in \mathbb{N} \end{aligned}$$

Therefore, consequential errors, caused by faulty predictions in early timesteps, can be prevented, which improves training time, especially in early epochs.

To project words and characters onto a vector representation, the sequence-to-sequence model uses an embedding mechanism. In case of word-level tokens, each word in the vocabulary is given a natural number index. Some special tokens have reserved indices, including the beginning-of-sequence and end-of-sequence tokens, a token for unknown words, and a padding token, used to pad sentences with different lengths into a batch. As part of a preprocessing step, all sentence tokens are then replaced by their respective indices. These indices can now be transformed into one-hot encodings, that can then be fed into a regular, fully-connected layer.

$$onehot(i) = \begin{pmatrix} onehot(i)_1 \\ \vdots \\ onehot(i)_{i-1} \\ onehot(i)_i \\ onehot(i)_{i+1} \\ \vdots \\ onehot(i)_V \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$
$$\text{vocabulary size } V \in \mathbb{N},\ i \in \mathbb{N}_{\leq V}$$

Since the activations of fully-connected layers are calculated via matrix multiplication, which is a rather expensive operation complexity-wise, many frameworks offer dedicated embedding layers. Working directly with the token indices, an embedding layer extracts row vectors of the weight matrix by using a computationally cheaper operation, like the gather operation, implemented in the PyTorch framework [9].

Word-based vocabularies are usually large, based on the number of different words in the dataset. This results in high-dimensionality vectors when not using an embedding layer or when predicting word probabilities, e.g. in an ASR or machine translation (MT) system. Processing sentences on a character-level drastically shrinks the vocabulary size. Models that output character or word sequences, usually do so by outputting a probability distribution over the vocabulary at each timestep. This can be achieved by applying the softmax function to the last layer. The concrete token, or rather vocabulary index of the token, can be sampled by taking the argmax function of the probability vector.

During experiments on their sequence-to-sequence architecture, Sutskever et al. [34] found an increase in model performance when reversing the input sequence. That way, the first token of the input is temporally closer to the first token of the output. This raises the question, whether the input encoding vector really contains all information needed by the decoder. For a translation system, reversing input sentences might be a good engineering trick if the source and target language are similiar. English and German both have a similiar word order, so it is likely that the first words in the input and output sentence correspond to each other. However, when translating English to Japanese or other languages with a different word order, this trick could actually influence the translation quality for the worse. A solution, that captures the intuition of looking at the right place at the right time, is needed.

This solution comes in the form of the attention mechanism [38]. Instead of relying on a single hidden encoder state, an attention model builds a context vector at each decoder timestep. The context vector is a weighted sum of encoder states, with weights produced by the attention mechanism, allowing the decoder to pay attention to one or more encoder states at any timestep. Attention has been successfully applied in combination with RNN-based encoder-decoder architectures for MT [34] and for image captioning [41]. Recently, a sequence-to-sequence model completely devoid of recurrent layers has achieved competitive results in various tasks. The Transformer model by Vaswani et al. [38] uses several attention layers in a encoder-decoder-like architecture. Instead of recurrence, it encodes relative token positions with a positional encoding mechanism. The model was evaluated on a English-German and English-French translation task, and achieved the highest BLEU score amongst state-of-the-art neural MT systems, most of them using recurrent and/or convolutional layers.

2.2.8 Hyperparameters

Training neural networks requires a lot of fine-tuning. Making the right design choices is paramount for solving difficult tasks, these choices include:

- Layer types (fully-connected, convolutional, recurrent, etc.)
- Number of layers and neurons per layer
- Batch size
- Learning rate and optimization algorithm (stochastic gradient descent (SGD), RMSProp, Adam, etc.)
- Activation functions (sigmoid, tangens hyperbolicus, ReLU, etc.)
- Loss function (MSE, cross-entropy, L1/L2 regularizations, etc.)
- Intermediate layer mechanisms (attention, batch regularization, gradient clipping, etc.)

These are called hyperparameters, as they are different from the trainable parameters in a neural network, because they can't be learned by gradient descent. Optimizing hyperparameters involves engineering some of them by hand, based on common practices and experiences, while others can be optimized with a random search algorithm.

Some common problems in neural network training have already been mentioned, like the vanishing and exploding gradient problems 2.2.5. Another common problem stems from the lack of sufficient training data for many real-world applications. Consider a network architecture with thousands of trainable parameters, but a relatively small number of training samples. The network is optimized by minimizing its loss function, but it simply memorizes all of the training samples, by encoding them in its abundant number of parameters. That leads to a training loss of zero or nearly zero, but the test loss will be significantly higher, as the network didn't actually learn higher-order patterns, but simply memorized training data. This phenomenon is called overfitting [12], and is illustrated in figure 2.8. It causes the network to generalize bad, as it can only produce correct output for training samples.

In practice, acquiring more data is often not a feasible solution for overfitting networks, as annotated data often requires human interaction and is therefore expensive. Partial solutions to this problem involve decreasing the network complexity, to weaken its memorization ability, and force it to properly learn patterns in order to minimize the loss. A relatively easy approach is the reduction of network parameters, by removing layers and decreasing the number of neurons for each layer. Additionally, data augmentation can help increasing the ammount of available data, but only to a certain degree.

When the network complexity cannot be reduced further without impairing the predictive abilities of the model too much, another approach is the use of dropout layers [33]. Dropout layers randomly disable neurons during training, by setting their activations to zero. The number of disabled neurons is determined by the dropout factor $p \in [0, 1]$, that represent the probability of each neuron to be deactivated. This method hinders the model to exactly encode training samples in its neurons, and additionally makes it more robust in general. Since the neurons learn to adjust their activations during training time, in order to compensate the missing neurons, the neuron output has to be scaled according to p during inference.

The inverse problem to overfitting is called underfitting [12], also depicted in figure 2.8. It occurs when a model does not have enough parameters to learn general patterns, or to

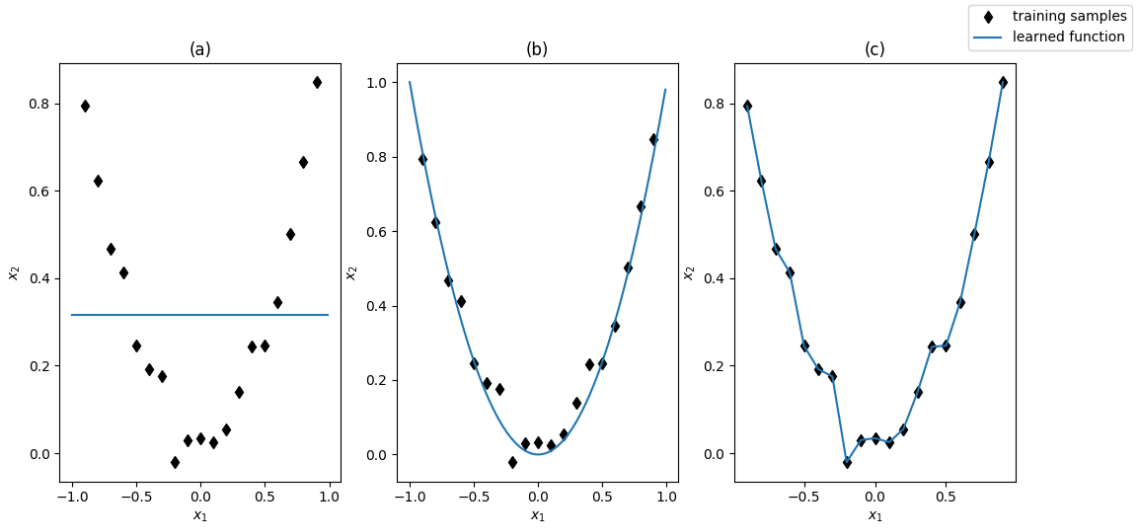

Figure 2.8: Regression example of (a) underfitting, (b) good generalization, (c) overfitting

memorize the training data. Underfitting can be detected when both training and test loss are oscillating instead of decreasing. Often it helps to simply add layers and neurons until the training loss shows a more natural behavior. A dimensionality reduction method 2.2.1 can also help by decreasing the input complexity, removing unnecessary information from the data and thereby simplifying it.

Chapter 3

Related Work

This chapter presents the foundation of this theses, the Speech Chain framework, as well as related approaches to quality estimation of text transcriptions.

3.1 Speech Chain

The Machine Speech Chain presented by Tjandra et al. [37] is a closed-loop architecture for simultaneously training an ASR and a TTS component. Its design is inspired by the speech chain with auditory feedback described by Denes et al. [10], that plays a crucial role in the learning process of humans. In comparison to the individual training of ASR/TTS systems, the advantage of the Speech Chain framework is its ability to train the systems using both labeled and unlabeled data.

The following describes the Speech Chain implementation for a single-speaker recognition task, as it is the same one used in my own experiments 5.

3.1.1 Training Process

A Speech Chain training iteration involves a supervised and an unsupervised training step. During the supervised training step, both the ASR and the TTS generate their hypotheses using teacher-forcing with the labeled data. The loss of both systems is calculated and then weighted by a scalar factor $\alpha \in \mathbb{R}_{\geq 0}$, resulting in $loss_{paired} = \alpha * (loss_{ASR,paired} + loss_{TTS,paired})$.

In the unsupervised training step, an unpaired data sample is used, meaning a sample without corresponding reference. One of the systems is used as an encoder and generates a hypothesis transcription/speech waveform. This hypothesis is then utilized as the input for the second system, used as a decoder, generating speech/text by again using teacher-forcing with the encoder's hypothesis, thereby resembling an autoencoder-like architecture. The decoder loss can then be calculated by treating the original data sample as the reference. For clarification, the unpaired TTS loss is calculated by generating a text transcription of an unpaired speech sample using the ASR, and then generating a speech waveform by using that transcription as an input for the TTS. On the other hand, the unpaired ASR

loss is calculated by generating a speech waveform of an unpaired text sample using the TTS, and then generating a transcription by using said waveform as an input for the ASR. This process is also shown in figure 3.1. Analogically to the supervised step, the losses are weighted by a $\beta \in \mathbb{R}_{\geq 0}$, to form $loss_{unpaired} = \beta * (loss_{ASR,unpaired} + loss_{TTS,unpaired})$.

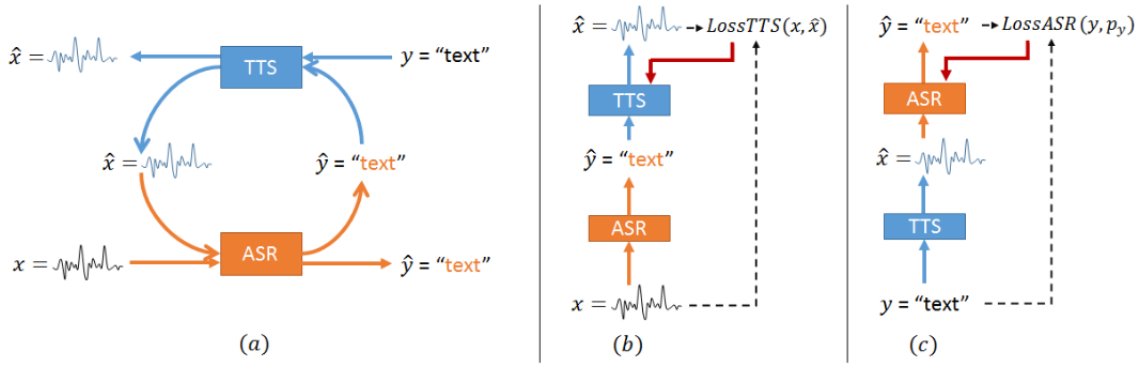

Figure 3.1: (a) Speech Chain architecture, (b) ASR to TTS iteration, (c) TTS to ASR iteration [37]

The total loss used to calculate the parameter gradients and update those parameters, is $loss = loss_{paired} + loss_{unpaired}$. Since the paired loss is more accurate due to its guaranteed correct label, α is usually larger than β. To improve the performance and convergence speed of the Speech Chain significantly, both ASR and TTS are usually pretrained, using only the labeled part of the dataset. This is equivalent to setting β to 0 until both ASR and TTS stop improving, before using a $\beta > 0$.

3.1.2 ASR Model

A neural sequence-to-sequence model is used for the ASR architecture. Its input is formed by a sequence of log Mel scale spectogram feature vectors. This neural network is used to directly model the posterior probability $p(y|x)$ of the character sequence $y = (y_1, \ldots, y_T)$ given the speech feature sequence $x = (x_1, \ldots, x_S)$. By applying the argmax function to $p(y|x)$, a hypothesis transcription can be generated.

The model is built using an encoder-decoder architecture with an attention module, as seen in figure 3.2. The feature vectors x are given to the encoder, which produces a representation in the form of hidden states h_S^e. To now generate the model's output, a starting token y_{start} is given to the decoder, to produce y_1. This step is repeated with y_1, producing y_2, and so forth, until reaching a fixed limit or an end-of-sequence token is predicted. The attention is used by the decoder to highlight important information stored in the encoder, based on the current decoder hidden states.

Given the reference labels $y \in \{1, \ldots, C\}^T$, which correspond to character indices of C transcribable characters, and the predicted posterior probabilities $p_{y_t} \in [0, 1]^C$, the loss can be calculated as follows:

$$loss_{ASR}(y, p_y) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{c=1}^{C} \mathbb{1}[y_t = c] * \log p_{y_t c}$$

3.1.3 TTS Model

Similiar to the ASR model, the TTS is also a sequence-to-sequence neural network modeling $p(x|y)$, for the speech waveform x given the character sequence y. It can be seen as an inverse model to the ASR 3.2.

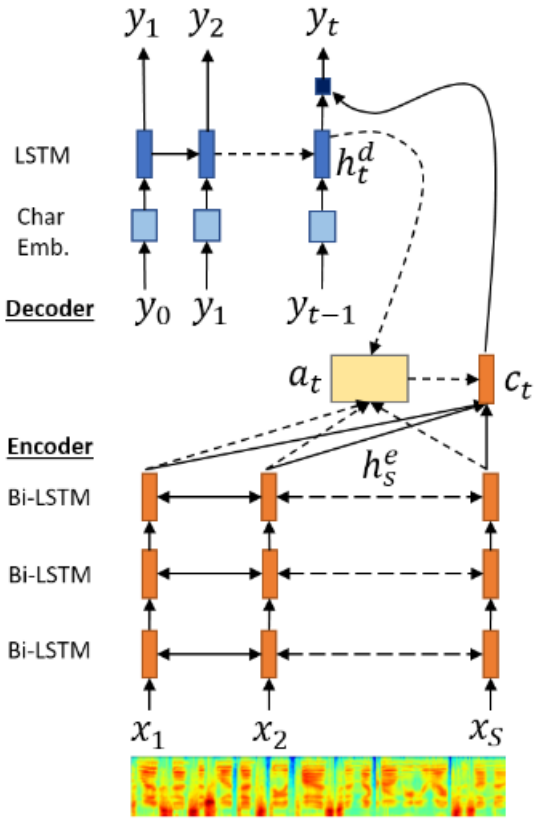

Figure 3.2: ASR encoder (orange) and decoder (blue), as well as attention module (light orange) [37]

The architecture of the TTS is based on the Tacotron [39] model. The character sequence is encoded in the encoder by using a character embedding layer, and processed by several fully-connected layers and a CBHG block. Also similiar to the ASR, the TTS hypothesis is generated by feeding the previous output into the decoder at each time step. These outputs x^M are a sequence of log Mel scale spectogram frames. Additionally, the TTS model also features a second output layer, consisting of another CBHG block and a fully-connected layer. After the full x^M was generated, the second output layer is used to predict the log magnitude spectogram x^R. A third output layer predicts b_s, which models whether the current frame x_s^M is the last frame in the sequence.

The loss function used to train the model is based on the MSE. The predicted outputs are denoted with $\hat{x}^M, \hat{x}^R, \hat{b}$, the ground truth labels with x^M, x^R, b.

$$loss_{TTS}(x, b, \hat{x}, \hat{b}) = \frac{1}{S}\sum_{s=1}^{S}(x_s^M - \hat{x}_s^M)^2 + (x_s^R - \hat{x}_s^R)^2 \\ -(b_s \log(\hat{b}_s) + (1 - b_s)\log(1 - \hat{b}_s))$$

3.1.4 Experimental results

The experiments of Tjandra et al. [37] compared an ASR and a TTS trained on labeled data, before and after applying the Speech Chain algorithm. For the single-speaker dataset, Google TTS was used to generate speech from the BTEC corpus [18]. The data is split into 10.000 paired utterances (20%) and 40.000 unpaired (40%). The experimental results can be seen in table 3.1.

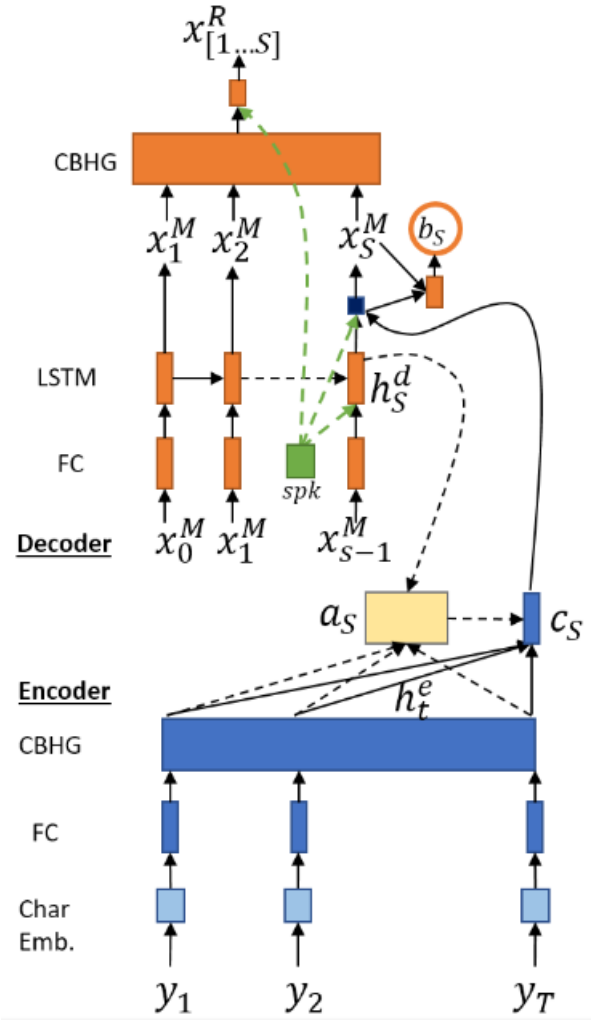

Figure 3.3: Tacotron encoder (blue) and decoder (orange), as well as attention module (light orange) and speaker embedding (green) [37]

Hypotheses were generated by using beam search. An especially strong performance increase can be observed for the ASR, where the CER has an absolute decrease of more than 4%. The subjective evaluation of TTS-synthesized speech also showed an improvement for the TTS.

3.1.5 End-to-End Loss via Straight-Through Estimator

As a follow-up publication to the original Speech Chain paper, Tjandra et al. [35] proposed an approach to additionally backpropagate the TTS loss through the ASR during the unsupervised ASR to TTS iteration. This is shown in figure 3.4, in contrast to figure 3.1. They tackled the problem of the non-differentiability of the argmax function during generation of the ASR hypothesis, by replacing the softmax function with the Gumbel-softmax function [21] to sample the discrete transcription. Therefore, they were able to calculate an end-to-end loss through both TTS and ASR.

The loss function used to optimize the ASR was a combination of a regular ASR loss and a TTS reconstruction loss. In order for that ASR loss to be calculated, the ASR hypothesis was generated by using teacher-forcing, similiar to the supervised training step. However, this requires a reference transcription for the ASR input, meaning this concept only works on paired data. In the context of this thesis, the data in the unsupervised training step is

data	(α, β)	ASR (CER)	TTS (MSE)
paired	-	10.06%	0.938
paired + unpaired	$(1, 0.25)$	5.44%	0.844

Table 3.1: Speech Chain experiment results comparing ASR/TTS trained on labeled data (without Speech Chain loop), and on labeled + unlabeled data (with Speech Chain loop)

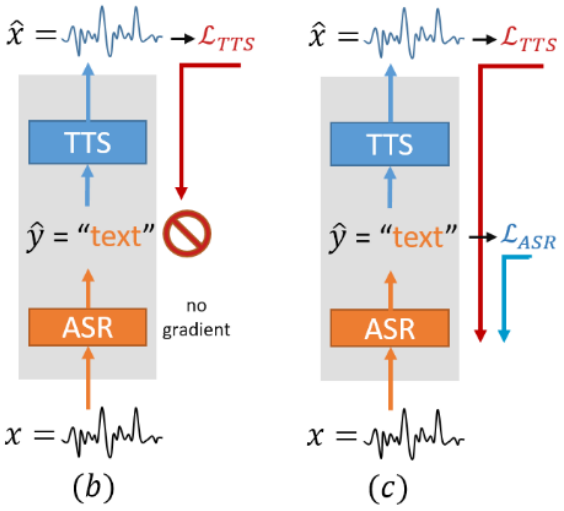

Figure 3.4: (b) original ASR to TTS iteration, (c) modified iteration with end-to-end loss [35]

treated as unpaired. For that reason, the Gumbel-softmax straight-through estimator is not applicable.

3.2 Quality Estimation

Quality estimation in the context of speech is already well known and used in modern MT systems. It is applicable in cases where there is no reference translation available. QE tasks were repeatedly among the various tasks of the WMT [4]. At first, QE approaches in MT where used to estimate metrics, such as BLEU score [28] and WER. However, these metrics are difficult to interpret at the sentence level, as local irregularities can get neglected in long utterances. For that reason, a better definition of quality is needed. This definition is not always transferable to other tasks, as it depends on the concrete use case of the QE component. QE can be used for highlighting segments that need post-editing, selecting alternate translations produced by different systems, and various other tasks in MT. In recent years, it has also gained increasing significance in ASR systems, where it can be used to detect whether an utterance was transcripted correctly, or which transcription is the best out of multiple alternate transcriptions.

3.2.1 Automatic Speech Recognition

QE systems designed for MT, like the prominent QuEst framework [32], are not applicable for the Speech Chain, since they are built to estimate translation qualities, not transcription qualities. In ASR, a common metric are confidence estimates. It is difficult to draw a definite border between quality estimation and confidence estimation, as the terms are sometimes used interchangebly in existing literature. For the context of this work, confidence is defined as a representation of the (internal) certainity of the ASR system about

its output. Quality on the other hand, can also rely on external factors, not known to the ASR at runtime.

Many QE systems for ASR are intended for the use in traditional HMM-GMM models, like the system presented by Negri et al. [26]. They mainly use features extracted from the acoustic model, word graphs or the internal language model. With the sequence-to-sequence ASR model present in the Speech Chain, these kinds of features cannot be extracted.

Other related works restrict themselves to only using black-box features, e.g. TranscRater [20]. This allows them to be independent from the actual system, and to be used across multiple ASR systems. As the QE component in this work is only employed in the Speech Chain, this kind of restriction is not needed, and thus white-box features are viable to use as well.

Fan et al. [11] present an approach for WER prediction in their work. This is an interesting take, as it could allow a more flexible use of the QE component in the Speech Chain, e.g. by dynamically adapting the quality threshold during runtime. However, they require a specific ASR architecture, and with the current, binary quality definition used in this work, it is unnecessarily complex to regress the WER and therefore not needed.

Chapter 4

Method

My modifications to the Speech Chain loop are separated into two parts. First, I design and implement a quality estimation component, that aims to measure the quality of ASR transcriptions. This component then decides whether such a transcription is good enough to be passed to the TTS and perform an optimization step on it, since it was generated unsupervised and does not have a corresponding reference. The second modification covers the concept of an input/output layer sharing mechanism between ASR and TTS, to further tighten the connection between both systems.

4.1 ASR Quality Estimation

While further processing recognized speech, it is often important to decide whether that speech was recognized correctly. Humans are able to assess the quality of their recognized speech by using various factors: The transmission medium, accent of the speaker, background noise and various other parameters can be perceived by the listener. Combining this knowledge with what was understood, one can reconsider if the speech was clearly understandable. If not, it might be necessary to ask for a repetition, e.g. in case of a conversation, or to just drop the information.

ASR systems do not possess this innate ability to assess their own quality. They are, however, able to produce metrics such as confidence scores, that can be used for QE. In machine translation tasks, QE systems are already commonly used to find segments that need manual post-editing or to choose between multiple alternative translations. For the Speech Chain framework, the QE component has a different use: By identifying bad hypotheses produced by the ASR, the unsupervised training step can be interrupted early. The advantage of this new capability becomes clearer in the following scenario:

Consider an incorrect ASR transcription t^{hyp} of the reference speech s^{ref}. Since this training step is unsupervised, the comparatively high error of the ASR is unknown to the Speech Chain. The TTS receives t^{hyp} as input and correctly produces speech s^{hyp}, and the resulting loss of this training step is calculated between s^{ref} and s^{hyp}. However, in this scenario, the loss will be quite high since s^{hyp} will not match s^{ref}, although the TTS

did a perfect job. During backpropagation, this can produce gradients that hinder the convergence of the TTS.

By applying QE, the incorrect t^{hyp} can be detected and the training step aborted, preventing the TTS from adapting to a non-representative loss. The originally proposed Speech Chain already implements a filter mechanism for bad hypotheses t^{hyp}, by canceling a training step if t^{hyp} does not terminate with an end-of-sequence token. All QE models presented in the following sections are built on top of the filter for unterminated hypotheses.

4.1.1 Classification vs. Regression

The goal of the new QE component is to estimate the quality of an ASR hypothesis t^{hyp}. In this work, quality is a binary value that indicates a sufficient similiarity of t^{hyp} to its reference t^{ref}. The ground-truth is obtained by comparing the CER of t^{hyp} regarding t^{ref} against a quality threshold θ.

$$quality_{ref}(t^{hyp}) := \begin{cases} 1, & CER(t^{hyp}, t^{ref}) \leq \theta \\ 0, & CER(t^{hyp}, t^{ref}) > \theta \end{cases}$$
$$\theta \in [0, \infty)$$

Now there are two possible approaches for the QE component: Either regress the CER of the hypothesis and compare the regression result against θ, or directly classify the hypothesis into one of the quality classes 0 and 1. Since there is no direct need to predict the CER, and, in my experiments, the former approach proved more difficult to build robust models with, I decided to build all following models as classifiers. A possible drawback of that decision is, that the QE component loses the capability to adapt θ during its deployment in the Speech Chain.

4.1.2 Baseline

To evaluate the competitivity of the proposed QE models, they are compared against a baseline estimator. This baseline estimator is designed around the correlation between the mean of the posterior probabilities of transcribed characters and the CER of a hypothesis. In this sense, the posterior probability mean should by higher for better hypotheses, and equal one in a perfect transcription. Based on this intuition, a first baseline estimator looks like this:

$$quality_{baseline}(t^{hyp}) = \begin{cases} 1, & 1 - s \leq \theta_{baseline} \\ 0, & 1 - s > \theta_{baseline} \end{cases}$$
$$s = \frac{1}{|t^{hyp}|} * \sum_{i=1}^{|t^{hyp}|} p_i$$
$$\text{posterior probabilities of } t^{hyp}\ p \in [0,1]^{|t^{hyp}|}$$
$$\text{parameter } \theta_{baseline} \in \mathbb{R}_{\geq 0}$$

p_i is the posterior probability of the i-th character in the transcription, given by the ASR. $\theta_{baseline}$ is a parameter similiar to θ, that must be optimized on a development set.

During early tests of the Speech Chain, the ASR showed a better performance for shorter utterances. Additionally, a larger percentage of character errors could be found in the latter

parts of a utterance. Following this observation, the baseline estimator can be improved by separating the hypotheses into n segments, and weighting these segments with different factors w_i.

$$quality'_{baseline}(t^{hyp}) = \begin{cases} 1, & 1 - s_{w,l} \leq \theta_{baseline} \\ 0, & 1 - s_{w,l} > \theta_{baseline} \end{cases}$$

$$s_{w,l} = \frac{1}{|t^{hyp}|} * \sum_{j=1}^{n} w_j \Big(\sum_{i=l_j}^{l_{j+1}} p_i \Big)$$

$$\text{posterior probabilities of } t^{hyp}\, p \in [0,1]^{|t^{hyp}|}$$

$$\text{parameters } \theta_{baseline} \in \mathbb{R}_{\geq 0},\, w \in \mathbb{R}^n_{\geq 0},\, l \in \{1, \ldots, |t^{hyp}|\}^{n+1}$$

The segment boundaries l_i, together with the segment weights w_i, are also to be optimized on a development set.

4.1.3 Manual Feature Extraction

When designing a model, there are two kinds of possible input features: Black-box features are independent from the inner workings of the ASR, and include features like the source Mel spectogram and the hypothesized transcription text. White-box features are typically specific to the ASR architecture. In case of the Speech Chain, since the ASR is a fully neural sequence-to-sequence model, useful white box features are output layer activations, the attention matrix of the decoder and the encoded speech representation.

Since this work only considers QE in the context of the Speech Chain, there is no necessity to restrict the features to the black box features. Instead, the QE can make use of any information that is available during the unsupervised training step.

One of the advantages of using deep neural network models is, that the model learns to extract certain information from the input by itself. Nevertheless, by manually creating some additional features that could have a good correlation with the prediction target, a model can concentrate on more important information.

For that reason I chose 11 "hand-crafted" features, that are extracted from the black and white box features in a preprocessing step. These features are:

- Number of characters in hypothesis
- Number of frames in source spectogram
- Posterior probability mean and standard deviation over the whole hypothesis
- Output layer activation entropy mean and standard deviation
- Attention matrix entropy mean and standard deviation
- Attention matrix monotonicity, measured by the inversion number of the argmax of the matrix rows
- Out-of-vocabulary rate, based on the vocabulary of the training corpus
- Language model score of the hypothesis in an independent language model

Through feature ablation, the number of hypothesis characters and source frames, together with the posterior probability mean and language model score, turned out to be the most predictive features. The other features had a smaller impact on the prediction result, but didn't influence it in a negative way.

4.1.4 MLP Model

To build an optimal classifier for the QE task, I built various neural network models. They all produce class probabilities $f(x)$ as an output, from which the quality can be inferred.

$$quality_{neural}(x) = \begin{cases} 1, & f(x)_1 \geq f(x)_2 \\ 0, & f(x)_1 < f(x)_2 \end{cases}$$
$$\text{quality estimator } f : X \longrightarrow [0,1]^2$$
$$\text{feature tensor } x \in X$$

The first model is a basic Multilayer Perceptron 4.1. It is trained on the manually extracted features described in 4.1.3. The model features several hidden layers and a two-dimensional output layer. To produce class probabilities as an output, the last layer includes the softmax function.

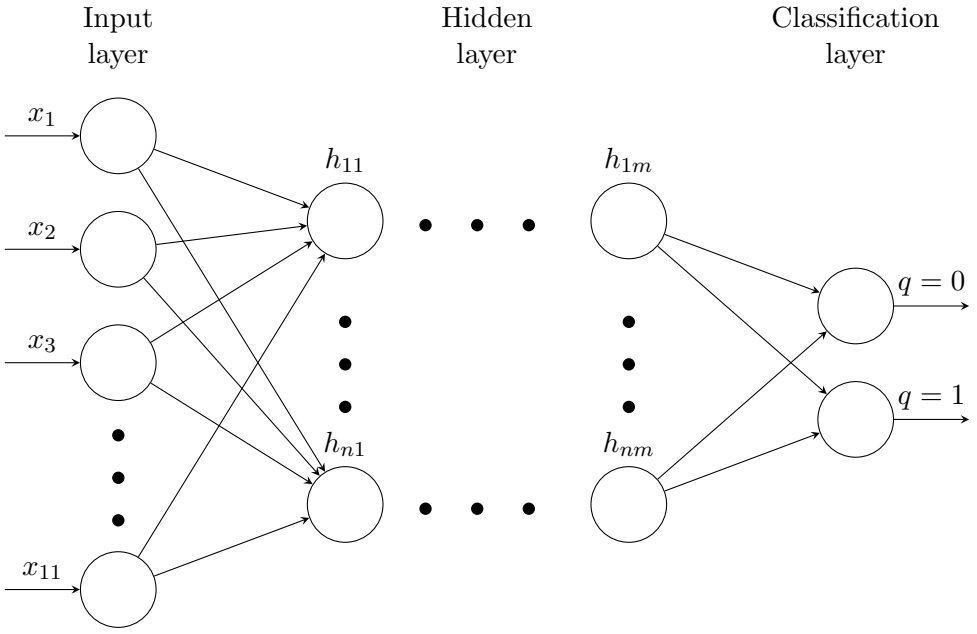

Figure 4.1: MLP quality estimator

The downside to this model is, that it only relies on the hand-crafted features. Those only contain information over the whole utterance, so local particularities in the data can get compensated by the rest. This can especially be observed in long utterances.

4.1.5 Recurrent Model

To capture these local particularities, the model needs to handle sequential input. For that reason, a bidirectional long short-term memory (Bi-LSTM) network 4.2 is used to encode the hidden states of the ASR decoder. This enables the model to use the context of the whole hypothesis through the bidirectional, recurrent connections. The classification is done by taking a fixed-size representation of the encoded input, and feeding it into a fully-connected layer.

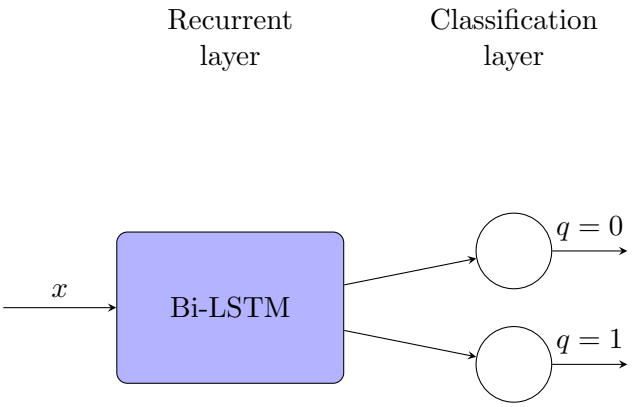

Figure 4.2: Quality estimator with recurrent layer

4.1.6 Multimodal Model

While the recurrent model presented in section 4.1.5 can process entire sequences, it doesn't use information apart from the ASR output, like the model 4.1.4 does. To utilize all of this information, I propose a third, multimodal neural network model 4.3. It processes the ASR decoder hidden states and the hand-crafted features individually at first, and then stacks and processes them together in several hidden layers.

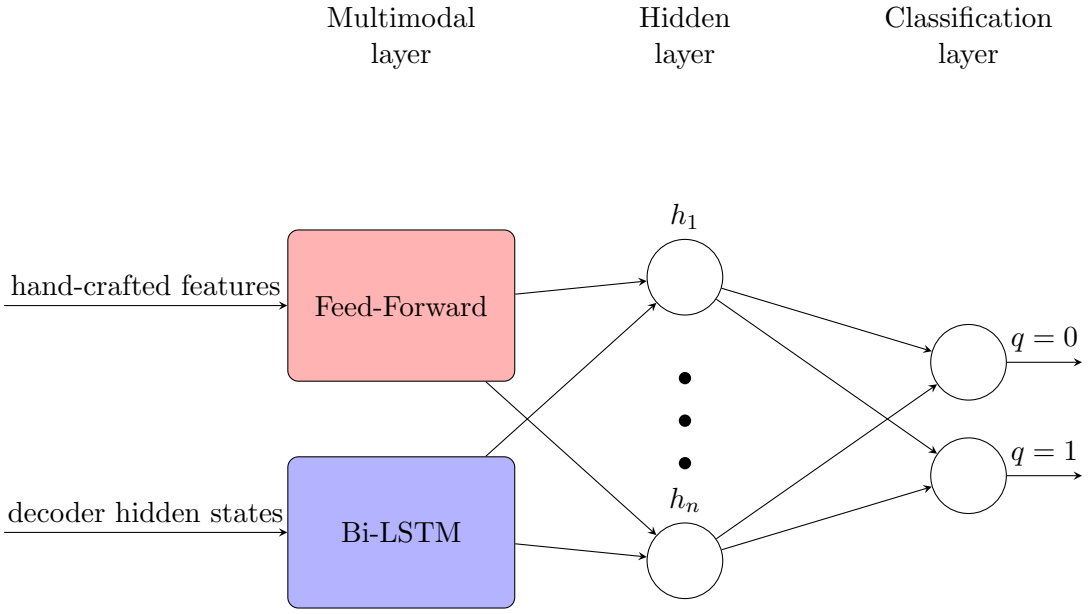

Figure 4.3: Multimodal quality estimator, combining the MLP and recurrent models

4.2 Input/Output Layer Sharing

In their publication, Tjandra et al. [37] successfully applied the mutual influence of speech production and recognition to ASR and TTS systems resulting in a semi-supervised deep learning framework. The concept, that is motivated from natural speech, was first introduced by Denes et al. [10], and explains the crucial role of hearing one's own speech in the healthy development of speaking capabilities. This so called auditory feedback is also a reason why people that suffer from deafness also suffer from deterioration of their speech.

To incorporate these two mechanisms even further into the Speech Chain framework, I propose a method that enables the ASR and TTS systems to directly learn from each

others mistakes. This can be seen as an analogy to the component connecting the speech production and recognition organs in the human body, which is the brain. This connection is established by redirecting the output of one system as input to the other system, thus sharing their input and output layers. Hereby, two advantages are created: First, the TTS can see the full output distribution of the ASR hypothesis, something that could be useful in an early iteration of the Speech Chain, where the ASR is still produces uncertain output. Second, a full backpropagation is achieved, so that the TTS loss gradient with respect to the weights of the ASR can be calculated, and vice versa. The second advantage can potentially become even more useful when working together with the QE component, as the QE can prevent bad ASR transcriptions from producing gradients that influence the systems in a negative way.

The modifications necessary to obtain this input/output layer sharing are described in the following.

4.2.1 ASR to TTS Iteration

Originally, in the unsupervised ASR to TTS training step, the ASR generates a hypothesis t^{hyp} by applying the argmax function to its output layer x^{out}. The size of the output layer is the same as the vocabulary size v. The generated sequence is then passed to the TTS.

$$\begin{aligned} t_i^{hyp} &= \underset{1 \leq j \leq v}{\operatorname{argmax}} x_{ij}^{out} \\ \forall i &\in \{1, \ldots, |t^{hyp}|\}, \\ x^{out} \in \mathbb{R}^{|t^{hyp}| \times v},\ t^{hyp} &\in \{1, \ldots, v\}^{|t^{hyp}|} \end{aligned}$$

With this approach, it is impossible to backpropagate the resulting TTS loss to the ASR, since the argmax function is not differentiable. To circumvent this problem, the softmax function is instead applied to the ASR's output layer, resulting in a vector of posterior probabilities. Therefore the output sequence $t^{hyp,\prime}$ is no longer a sequence of vocabulary indices, but a sequence of these posterior vectors.

$$\begin{aligned} t_i^{hyp,\prime} &= softmax(x_i^{out}) \\ \forall i &\in \{1, \ldots, |t^{hyp}|\}, \\ t^{hyp,\prime} &\in [0,1]^{|t^{hyp}| \times v} \end{aligned}$$

Since the softmax function is differentiable, the TTS loss gradient can now be calculated with respect to the ASR weights. However, the TTS needs to be adapted to meet the new input format requirements.

4.2.2 TTS Modification

The TTS model used in the originally proposed Speech Chain is a variation of the Tacotron model. [39] Its input layer features an embedding layer, that transforms the input sentence into an embedding for the encoder to use. The input sentence is represented as a sequence of vocabulary indices, while the vocabulary consists of alphabetical character tokens, as well as special tokens for beginning-of-sentence, end-of-sentence, space, punctuation, etc.

With the modification of $t^{hyp,\prime}$, a character embedding layer is not applicable anymore, since the input does not consist of vocabulary indices. Instead, it is replaced with a fully-connected layer. The key differences between those layers are largely dependent on the

implementation. In PyTorch for example, character embedding layers are implemented without biases and use a gather operation instead of a costly matrix multiplication to calculate the layer activations [9].

The modified TTS can now be pretrained by transforming vocabulary indices into one-hot representations. For one-hot encoded input, a fully-connected layer without biases behaves the same way as the embedding layer does. An one-hot vector v with $v_i = 1$ can be viewed as an equivalent to a probability vector that represents a probability of 100% for the i-th character in the vocabulary. This concept can be expanded to arbitrary probabilities, and will become useful in the unsupervised training step, where the posterior vectors produced by the ASR are fed into the TTS.

One possible drawback of this method is, that the TTS only sees one-hot vectors during pretraining, which could make the system react too sensitive when it first gets input other than one-hot encodings. In order to adjust the TTS as early as possible to this more "chaotic" type of input, I propose introducing random noise to one-hot vectors during supervised training steps (including pretraining).

$$noisyonehot(i) = onehot(i) + noise(i) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} c_1 \\ \vdots \\ c_{i-1} \\ -\|c\|_1 \\ c_{i+1} \\ \vdots \\ c_n \end{pmatrix}$$
$$c \sim Dist(\dots)^n$$

The noise c is sampled from a probability distribution $Dist$ with parameters, that are inferred from posterior vectors produced by a pretrained ASR. This way, the amount of noise in the supervised training steps is similiar to that during unsupervised training. The "noisy" one-hot encoding is constructed so that $\|noisyonehot(i)\|_1 = \|onehot(i)\|_1 = 1$, this also holds true for probability vectors produced by the softmax function.

4.2.3 TTS to ASR Iteration

Input/output layer sharing from TTS to ASR is easier to implement, as the TTS hypothesis is already differentiable. Since the ASR loss in the Speech Chain is calculated before applying the argmax function, that loss can be backpropagated not only through the ASR, but the TTS as well.

Chapter 5

Experiments

To evaluate my proposed method, experiments are conducted on different combinations of the aformentioned modifications. The first experiment compares different quality estimator models to each other, independent from the Speech Chain. The second experiment includes the input/output layer sharing in one or both of the iteration cycles, so that both methods are brought together to evaluate the total improvement on the learning process.

5.1 Dataset

For the experiments, the LJ Speech dataset [19] is used. It consists of 13.100 utterances with a length of up to ten seconds. All texts are read by a single speaker, this matches the first Speech Chain experiment, in which a single speaker dataset was used as well. The audio clips were recorded by the LibriVox project [25] and have the advantage of being nearly free from background noise.

For training the systems, the utterances are split into sets of training, test and development data. The test and development sets each contain 3% of all utterances. The remaining utterances form the training set, which is further partitioned into supervised and unsupervised training data. During the training of the Speech Chain, the unsupervised training set is then assumed to not have paired text/speech, to simulate the training on audio data without transcriptions, or on text data without speech. The partitions consist of 25% paired and 75% unpaired data. All sets of utterances are pairwise disjoint sets.

Preparing a dataset for the quality estimator models poses a more difficult challenge. Training the QE models on a separate dataset would go against the idea of the Speech Chain, as the same data could be used in the supervised training of ASR and TTS, resulting in better Speech Chain components, even without any QE. For that reason, the QE models should make use of the same training data as ASR/TTS. The QE training data consists of hypotheses, generated by the ASR, using its training data. However, an ASR system after pretraining is not eligible for this generation task, as the performance on its own training data would be ”too good”, and nearly all of the hypotheses would have a CER close to zero, resulting in an unbalanced QE training set. As a compromise, ASR systems

from 5 different epochs of pretraining are chosen, namely from epochs 30, 35, 40 and 120, resulting in a balanced QE training set with 54.83% good and 45.17% bad quality samples.

5.2 Hyperparameters

Training the Speech Chain requires tuning lots of hyperparameters. QE hyperparameters include the quality thresholds θ and $\theta_{baseline}$ as described in sections 4.1.1 and 4.1.2, as well all design choices of the neural network models. Most of the parameters for the original Speech Chain are taken from the publication of Tjandra et al. [36].

5.2.1 QE Hyperparameters

The quality threshold θ is chosen by looking at the dataset and choosing it manually, since the parameter is rather subjective. For my experiments, I decided to use $\theta = 0.14$. $\theta_{baseline}$, together with the segment weights and boundaries w, l, was optimized on a development set by applying the improved baseline estimator and optimizing for accuracy. The optimiziation is done by using the COBYLA method as implemented in the SciPy package [8]. As a result, the following hyperparameters were chosen: $\theta_{baseline} = 0.15$, $w = (1.26, 0.92, 1.03, 0.92)$, $l = (0, 0.16, 0.46, 0.57, 1)$, with which the baseline estimator achieved an accuracy of 74.64% on a development set.

The QE models share some design choices, e.g. the loss functions. Because of the classification task and the two-dimensional output of the model, the cross-entropy function is a natural fit. As an optimizer, Adam proved to be the most reliable, with an initial learning rate of 10^{-4} and an L2 regularization factor of 10^{-5}. To achieve a better convergence, a learning rate scheduler was used in the experiments. That way, the learning rate got halved every 15 epochs after no decrease of the development loss. The sigmoid function was used as the activation function. To encourage better generalization, a dropout factor of 0.3 was used.

Further hyperparameter optimization for the QE models was done by using a random search approach. That way, the basic MLP model 4.1.4 was implemented with three hidden layers and 72 neurons each. The recurrent model seen in the tests has one Bi-LSTM cell with 116 neurons for each direction, and one hidden layer with 116 neurons. As a combination of the two aformentioned models, the multimodal QE model uses one hidden layer with 72 neurons for the feed-forward part of the network (see 4.3), and its Bi-LSTM has the same properties as the recurrent model. Both inputs are stacked and fed into a 188 neurons large hidden layer afterwards.

5.2.2 Speech Chain Hyperparameters

For the paired/unpaired loss coefficients α, β, $\alpha = 1$ and $\beta = 0.25$ are chosen, so that $loss = \alpha * (loss_{ASR,paired} + loss_{TTS,paired}) + \beta * (loss_{ASR,unpaired} + loss_{TTS,unpaired})$. The ASR predictions were generated via beam search with a beam width of 3. Initial learning rates were $2.5 * 10^{-4}$ for the TTS and 10^{-3} for the ASR. A scheduler halved the learning rates, in case of no development loss decrease for 3 consecutive epochs. Both the ASR and TTS use dropout with factors 0.25 and 0.5 respectively. The gradient clipping employed in my experiments is more strict compared to the original Speech Chain experiments, due to the substantially longer backpropagation chain. Therefore, the ASR gradient norm clipping limit was set to 5, and 2 for the TTS.

New hyperparameters are needed for the noisy one-hot vectors in the supervised TTS training steps. To find the distribution and parameters for the noise, a pretrained ASR predicted character posterior probabilities p_i, and the amount of uncertainity $1 - \max_{i \in \{1,\dots,|p|\}} p_i$

was plotted in a histogram. Next, by fitting various probability distribution on the histogram, the alpha distribution [22] with parameters $\alpha_d = 1.55, loc = -0.03, scale = 0.13$ was chosen, so that the noise coefficient $\frac{c-loc}{scale} \sim AlphaDist(\alpha_d)$.

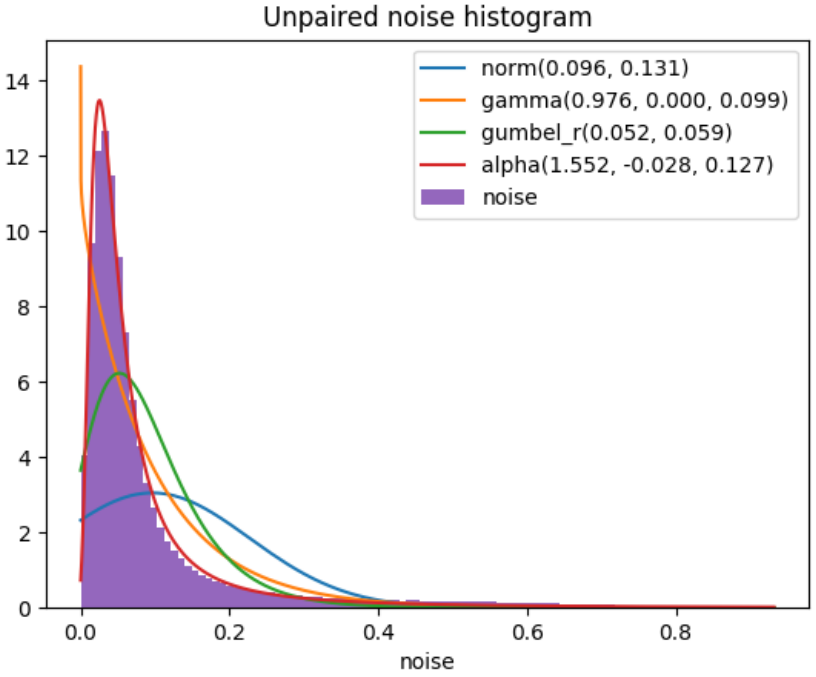

Figure 5.1: Noise histogram with various distributions, including Gaussian, gamma, Gumbel and alpha distributions

Fitting the distribution was done using the SciPy library's statistics package [8].

5.3 Results

The experiments were conducted in a two stage fashion. In the first stage, the different QE approaches are compared against each other, and afterwards the best quality estimator is used in the Speech Chain. The second stage involves testing the original Speech Chain against its modification with input/output layer sharing, and lastly the ASR QE in the loop.

5.3.1 QE Results

The compared QE models include the MLP model 4.1.4, recurrent model 4.1.5 and multimodal model 4.1.6. Additionally, the recurrent model using posterior probabilities instead of ASR decoder hidden states was also evaluated. The baseline for the experiments was introduced in section 4.1.2. In the table 5.1 below, the accuracy, F_1 score, precision and recall the models achieved on a test set is depicted, with the best values highlighted in each column.

It shows that the MLP model has the best performance in terms of accuracy, F_1 score and precision, while being outperformed in recall by the multimodal model. Also, both the MLP and the multimodal model show better results than the already strong baseline, in contrast to the recurrent models. The reason for that is likely a severe overfitting that appeared in the recurrent model. Despite efforts like L2 regularization, batch normalization, dropout factors between 0 and 0.6, and a large decrease in network complexity

model	accuracy	F_1 score	precision	recall
baseline	0.673	0.693	0.714	0.674
MLP	**0.855**	**0.877**	**0.823**	0.938
recurrent	0.636	0.710	0.631	0.813
recurrent (posterior probabilities)	0.580	0.606	0.623	0.591
multimodal	0.841	0.870	0.789	**0.969**

Table 5.1: QE model comparison

by trimming the number of Bi-LSTM neurons down to 32, the overfitting could not be mitigated while still upholding any noteworthy performance of the network.

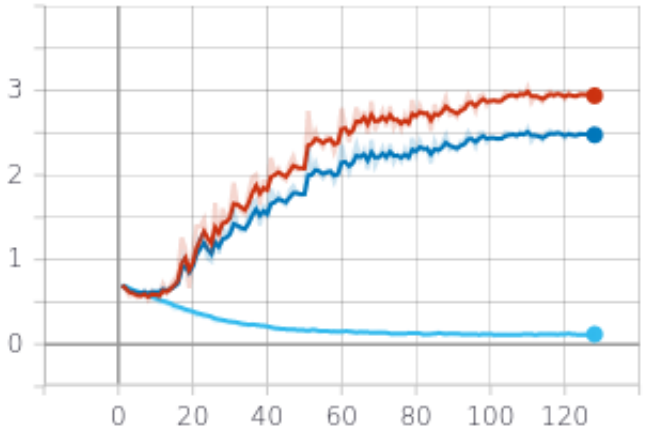

Figure 5.2: CE loss plot, training loss in light blue, test and development loss in red and blue

Based on the aforementioned experimental results, the MLP model is chosen to be incorporated into the Speech Chain.

5.3.2 Speech Chain Results

Before discussing the Speech Chain modification itself, let us take a look at the impact of the TTS modification 4.2.2. Because of the implementation-specific differences between character embedding layers and fully-connected layers, the modified TTS has more trainable parameters than the original, since the embedding layers does not have bias weights. The following table 5.2 compares the two TTS systems against each other based on their MSE on a test set. The experiments are conducted on the 25% split of the dataset, as described in 5.1, as well as the full LJSpeech corpus for comparison.

input layer	paired data	MSE
character embedding	25%	1.048
fully-connected without bias		1.045
fully-connected		**0.803**
character embedding	100%	0.670
fully-connected without bias		0.671
fully-connected		**0.650**

Table 5.2: TTS with character embedding input layer compared to fully-connected input layer

When looking at the 25% data experiments, it seems that the original TTS reaches a local optimum at a much earlier stage than the modification, and that local optimum prevents it from further improving. The TTS with the fully-connected input layer however, does not hit that local optimum and keeps decreasing its loss for 50 or more epochs after the original TTS finished. After conducting a subjective evaluation, it becomes clear that the modified TTS also produces much clearer speech than before. I hypothesize that the reason for this large difference in convergence is indeed the greater number of parameters in the modified TTS. To prove this, an additional TTS was trained 5.2, also with a fully-connected input layer, but without bias weights. This experiment turned out to produce speech similiar to the original TTS (with embedding layer). All hyperparameters stayed the same during all those experiments, so the additional bias is evidently the reason for the TTS improvement.

However, by looking at the experiments on 100% of the LJSpeech data, it becomes clear that this advantage is only relevant for small data experiments, as both TTS experiments turn out roughly the same when given the whole dataset to train on. By listening to the synthesized speech, there is little to no difference between both systems to be observed.

The final step is to piece the modified TTS and the QE together into a new Speech Chain loop variant. The experiments were conducted using input/output layer sharing in both directions, only from ASR to TTS, as well as only from TTS to ASR, and without any input/output layer sharing. All these experiments are done with and without QE in the loop. They can be compared by the CER of the ASR transcriptions, as well as the cross-entropy loss of the ASR and the MSE of the TTS on a test set. As a baseline, the original Speech Chain results can be found in the first row of the following table 5.3.

Speech Chain	QE	ASR (CER)	ASR (CE)	TTS (MSE)
baseline	no	5.408%	0.522	0.790
	yes	5.295%	0.518	0.855
no IOLS	no	**3.569%**	0.483	0.691
	yes	3.889%	**0.480**	0.694
ASR → TTS IOLS	no	3.816%	0.485	0.702
	yes	3.929%	0.486	0.701
TTS → ASR IOLS	no	4.417%	0.508	**0.689**
	yes	4.683%	0.503	0.696
bidirectional IOLS	no	4.301%	0.502	0.695
	yes	4.746%	0.510	0.692

Table 5.3: Speech Chain experiments with different IOLS/full backpropagation variants and with/without QE

One of the most apparent observations is the large drop in CER, and ASR/TTS loss from the baseline to all IOLS experiments. It is, however, highly likely that this fact solely comes from the better TTS, due to the input layer modification. To prove this, the no IOLS/no QE Speech Chain experiment was repeated with different ASR/TTS variants. In table 5.4, ASR systems with character posterior probability distribution outputs are compared against their discrete output counterpart, together with TTS input layer variations, to include or exclude bias parameters. The discrete ASR output is achieved by one-hot encoding the argmax of the output distribution. It is therefore similiar, but not identical to the original Speech Chain training process, as seen in the first row of table 5.3.

When comparing the different input/output layer sharing approaches, it seems like the TTS to ASR IOLS approach is most beneficial for the TTS. However, the difference in MSE is only small compared to other experiment results, especially to the experiment without IOLS. The subjective evaluation even shows, that the TTS to ASR input/output

ASR output	TTS input layer bias	ASR (CER)	ASR (CE)	TTS (MSE)
discrete	no	5.408%	0.522	0.790
	yes	4.105%	0.491	**0.683**
distribution	no	5.127%	0.513	0.801
	yes	**3.569%**	**0.483**	0.691

Table 5.4: Closer look at the impact of ASR output type and TTS input layer, the experiments are based on the Speech Chain without full backpropagation or QE

layer sharing produces less comprehensible speech than its counterpart. For the ASR, the best results are achieved by the Speech Chain without any IOLS.

The QE approach works well in theory, but has little effect on the experiments. Intuitively speaking, the quality classifier should start off restrictive and loosen up over time, becoming more and more irrelevant as the Speech Chain converges. This is actually the case: By observing the number of filtered ASR hypotheses in the Speech Chain experiment without IOLS, but with the QE component, it becomes evident that around 50% of unsupervised ASR to TTS iterations get interrupted due to QE in the first epoch. This also includes the hypotheses that are generated without an end-of-sequence tag, and get filtered anyway, even without the novel QE. That percentage decreases to 20% around epoch 6, and reaches 12% at epoch 74, the epoch with the best development loss. For comparison, the same experiment, but without QE, starts off with 20% invalid ASR transcriptions and only produces 9% invalid transcriptions at its last epoch, epoch 35. Nevertheless, the systems do not obviously benefit from that.

Both ASR and TTS loss differences between QE and no QE are in the expected range of training randomness. There is no clear pattern for when the QE improves one of the systems. Even the ASR without input/output layer sharing, but with QE, that features the lowest CE loss of all experiments, seems to be comparatively bad when looking at the CER of its transcriptions.

The only case in which QE improved both CER and CE loss, is when used in the baseline Speech Chain. Even so, the TTS in this case shows a large drop in performance. Since the QE only impacts the TTS in the first place, and the TTS gets worse, it is implied that either the ASR benefits from a bad TTS, which would go against the theory of Speech Chain, or the ASR improvement is within the range of randomness. This was confirmed by reconducting the experiment multiple times, where the test CER shows an oscillation of up to 0.2%.

Chapter 6

Conclusion

In this work, I discussed the possible application of an ASR QE component in the context of the Speech Chain. The QE component itself is built with recent machine learning methods, such as neural networks and LSTMs. It uses data that is specific to the ASR model present in the Speech Chain. After comparing different QE classifiers, the MLP model 4.1.4 outperformed the others, and reached an accuracy of 85.5% with a F_1 score of 0.877. This shows that QE is indeed practicable with the quality term defined in section 4.1.1, and the available data. Furthermore, a machine learning approach is suitable for this problem, as the chosen baseline achieves above average results, but still not as good as the neural network model, as seen in table 5.1.

After validating the viability of the QE component, it was implemented in the Speech Chain loop. Its purpose is to filter bad hypotheses from the ASR before they can be synthesized by the TTS. Additionally, to further tighten the training loop, a input/output layer sharing was implemented, that can be used together with the QE or on its own.

For the input/output layer sharing, a modification of the TTS' input layer was necessary, as the input shifted from a sequence of scalars to a sequence of vectors. The proposed fully-connected input layer included bias weights, contrary to the former character embedding layer. This modification alone was responsible for a much better convergence in the pretraining phase, which resulted in a lower test loss and arguably clearer speech, as seen in table 5.2 and the subjective evaluation. It was also shown, that the additional parameters introduced by the new input layer only affect the TTS in scenarios with a small amount of labeled training data.

With all these parts coming together, the Speech Chain was trained with all variants of input/output layer sharing, each with and without the QE component in the loop. The results in table 5.3 show, that while QE might work individually, it unfortunately did not improve the Speech Chain. The TTS seemed to improve even on the "bad" hypotheses, and filtering these hypotheses decreased the number of training samples. After all, the old machine learning wisdom applies here as well: "There is no data like more data."

The same goes for the input/output layer sharing, as none of the three experiment groups with IOLS performed better than the experiment without IOLS. Nevertheless, all experi-

ment achieved results better than the baseline, but it can be assumed that the better TTS is responsible for this improvement. Furthermore, the fact that the TTS can see the full output distribution of the ASR at runtime brought no significant advantage, as proven in table 5.4. I hypothesize that the gradients lose their context after the many layers the backpropagation algorithm has to cover, and therefore confuse the latter system.

6.1 Further Work

To conclude this thesis, I would like to talk about possible further work on QE in the Speech Chain. While the filtering of ASR hypotheses may not improve the Speech Chain performance, it is not implausible that filtering TTS hypotheses in the unsupervised TTS to ASR iteration might be more successful. This could be implemented by using a QE component that assesses the quality of TTS hypotheses.

Another possibility would be to change the way QE is applied in the Speech Chain. For example, there could be an additional term $loss_{QE}$ that could be added onto the loss function, or maybe a scalar multiplication factor that could be multiplied to the resulting gradients. This way, the QE could control the consequences of a possible bad hypothesis by weakening its impact on the systems. In order to achieve that, a more complex definition of quality, than the binary one presented in section 4.1.1 would be necessary, and the QE model would need to change its architecture accordigly.

Bibliography

[1] R. Bellman and Karreman Mathematics Research Collection. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961. URL https://books.google.de/books?id=POAmAAAAMAAJ.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[3] Hervé Bourlard and Nelson Morgan. *Hybrid HMM/ANN systems for speech recognition: Overview and new research directions*, pages 389–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-69752-7. doi: 10.1007/BFb0054006. URL https://doi.org/10.1007/BFb0054006.

[4] Chris Callison-Burch, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia. Findings of the 2012 workshop on statistical machine translation. 06 2012.

[5] Nancy Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th Conference on Message Understanding*, MUC4 '92, page 22–29, USA, 1992. Association for Computational Linguistics. ISBN 1558602739. doi: 10.3115/1072064.1072067. URL https://doi.org/10.3115/1072064.1072067.

[6] Chung-Cheng Chiu, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Katya Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017. URL http://arxiv.org/abs/1712.01769.

[7] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012. URL http://arxiv.org/abs/1202.2745.

[8] The SciPy community. Scipy reference guide - optimization and root finding, 2008. URL https://docs.scipy.org/doc/scipy/reference/optimize.html. [Online; accessed 20-April-2020].

[9] Torch Contributors. Pytorch master documentation - embedding, 2019. URL https://pytorch.org/docs/stable/nn.html#embedding. [Online; accessed 18-April-2020].

[10] P.B. Denes, P. Denes, and E. Pinson. *The Speech Chain*. Anchor books. Worth Publishers, 1993. ISBN 9780716723448. URL https://books.google.de/books?id=ZMTm3nlDfroC.

[11] Kai Fan, Jiayi Wang, Bo Li, Boxing Chen, and Niyu Ge. Neural zero-inflated quality estimation model for automatic speech recognition system, 10 2019.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Alex Graves, Santiago Fernandez, and Jürgen Schmidhuber. Bidirectional lstm networks for improved phoneme classification and recognition. pages 799–804, 01 2005.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[16] H. Hotelling. Analysis of a complex of statistical variables into principal components, 1933. URL `https://books.google.de/books?id=qJfXAAAAMAAJ`.

[17] Xuedong Huang, Alex Acero, Hsiao-Wuen Hon, and Raj Reddy. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall PTR, USA, 1st edition, 2001. ISBN 0130226165.

[18] Gen ichiro Kikui, Eiichiro Sumita, Toshiyuki Takezawa, and Seiichi Yamamoto. Creating corpora for speech-to-speech translation. In *INTERSPEECH*, 2003.

[19] Keith Ito. The lj speech dataset. `https://keithito.com/LJ-Speech-Dataset/`, 2017.

[20] Shahab Jalalvand, Matteo Negri, Marco Turchi, José de Souza, Daniele Falavigna, and Mohammed Qwaider. Transcrater: a tool for automatic speech recognition quality estimation. 01 2016. doi: 10.18653/v1/P16-4008.

[21] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2016. URL `http://arxiv.org/abs/1611.01144`. cite arxiv:1611.01144.

[22] Norman Lloyd. Johnson, Narayanaswamy Balakrishnan, and Samuel Kotz. *Continous univariate distributions*. Wiley, 1994.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, 1993.

[25] Hugh McGuire. Librivox project. `https://librivox.org/`, 2005.

[26] Matteo Negri, Marco Turchi, José Guilherme Camargo de Souza, and Daniele Falavigna. Quality estimation for automatic speech recognition. In *COLING*, 2014.

[27] Christopher Olah. Understand lstm networks, 2015. URL `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Online; accessed 17-April-2020].

[28] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL `https://doi.org/10.3115/1073083.1073135`.

[29] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[30] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[31] Apeksha Shewalkar, Deepika Nyavanandi, and Simone Ludwig. Performance evaluation of deep neural networks applied to speech recognition: Rnn, lstm and gru. *Journal of Artificial Intelligence and Soft Computing Research*, 9:235–245, 10 2019. doi: 10.2478/jaiscr-2019-0006.

[32] Lucia Specia, Kashif Shah, Jose Camargo, and Trevor Cohn. Quest - a translation quality estimation framework. 01 2013.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

[34] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL `http://arxiv.org/abs/1409.3215`.

[35] A. Tjandra, S. Sakti, and S. Nakamura. End-to-end feedback loss in speech chain framework via straight-through estimator. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6281–6285, 2019.

[36] A. Tjandra, S. Sakti, and S. Nakamura. Machine speech chain. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 28:976–989, 2020.

[37] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Listening while speaking: Speech chain by deep learning. pages 301–308, 12 2017. doi: 10.1109/ASRU.2017.8268950.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL `http://arxiv.org/abs/1706.03762`.

[39] Yuxuan Wang, R.j. Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, and et al. Tacotron: Towards end-to-end speech synthesis. *Interspeech 2017*, 2017. doi: 10.21437/interspeech.2017-1452.

[40] Wikipedia contributors. F1 score, 2006. URL `https://en.wikipedia.org/wiki/F1_score`. [Online; accessed 10-April-2020].

[41] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015. URL `http://arxiv.org/abs/1502.03044`.

Glossary

k-nearest neighbor algorithm classifies datapoints according to their k nearest neighbors

Adam RMSProp variant with momentum-carrying gradients

argmax function that returns the index of the largest element of an indexed collection (array, vector, tensor, etc.)

ASR automatic speech recognition

Bayesian statistics mathematical field based on a interpretation of probability, introduced by Reverend Thomas Bayes

Bi-LSTM bidirectional long short-term memory

BLEU bilingual evaluation understudy; algorithm to evaluate machine-translated text based on its similiarity to a professional human translation

CBHG convolution bank, highway network and bidirectional GRU; combination of multiple neural layers, including convolutional and (gated) recurrent layers, as well as residual connections

CE cross-entropy

CER character error rate

CNN convolutional neural network

COBYLA constrained optimization by linear approximation; numerical optimization method that does not rely on gradients, and therefore allows optimization of non-differentiable functions

computational linguistics science of modeling natural language

cross-entropy metric for the quality of a probability distribution, calculated as $H(P, Q) = -\sum_{x \in X} P(x) \log Q(x)$

data augmentation producing more features and data samples by varying existing data

expectation maximization algorithm that iteratively fits statistical model parameters to a set of data points

gather operation computationally fast extraction of column/row vectors of matrices

Google TTS TTS system developed and maintained by Google

Hadamard product pointwise multiplication of two vectors

HMM hidden Markov model

HMM-ANN model statistical model used to predict the probabilities of sequences, based on hidden Markov models and artificial neural networks

HMM-GMM model statistical model used to predict the probabilities of sequences, based on hidden Markov models and Gaussian mixture models

inversion number number of cases in which $\forall i, j \in \{1, \ldots, |s|\}, i < j : s_i > s_j$ for a sequence s

IOLS input/output layer sharing

LSTM long short-term memory

Markov chain stochastic process that models probability based on a fixed number of previous observations

Mel scale acoustic unit, describing the perceived pitch of sounds

MLP multilayer Perceptron

MSE mean squared error

MT machine translation

out-of-vocabulary rate percentage of words in a sentence that do not appear in a given vocabulary

PCA principle component analysis

phoneme small unit of natural speech that carries pronunciation information

posterior probability in the Bayes theorem, probability of a class given an observation

prior probability in the Bayes theorem, probability of a class without regarding the current observation, also called bias

prosody auditory features of spoken language not bound to phonemes, such as pitch, tone, rythm, intonation

PyTorch machine learning framework for Python, built on the Torch library written in Lua/C

QE quality estimation

rectified linear unit $ReLU(x) = \max\{0, x\}$

ReLU rectified linear unit

RMSProp SGD variant with parameter-individual learning rate adaption

RNN recurrent neural network

SciPy python library for numerical routines

SGD stochastic gradient descent

sign function $sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$

softmax also known as normalized exponential function, $softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

Speech Chain closed loop training architecture for an ASR and TTS introduced by Tjandra et al. [37], as well as human speech processing principle first described by Denes et al. [10]; in this work, the former definition is referenced if not stated otherwise

support vector machine linear classifier that aims to maximize the margin between classes

SVM support vector machine

TTS text-to-speech

Viterbi algorithm calculates the most likely state sequence in a HMM, that could produce a given observation

WER word error rate

WMT workshop on statistical machine translation