Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Faculty of Media Engineering and Technology
German University in Cairo
Trapeze Group Switzerland GmbH

Dispatcher Actions using Speech Recognition

Bachelor Thesis

Author:	Maggie Ezzat Gamil Gaid
Supervisors:	Dr. Karim Badawi, ETH Zürich and Trapeze Switzerland
	Dr. Mohamed Omar, Idea in Motion Egypt
	Prof. Dr. Slim Abdennadher, German University in Cairo
Submission Date:	29 August, 2019

Faculty of Media Engineering and Technology
German University in Cairo
Trapeze Group Switzerland GmbH

Dispatcher Actions using Speech Recognition

Bachelor Thesis

Author:	Maggie Ezzat Gamil Gaid
Supervisors:	Dr. Karim Badawi, ETH Zürich and Trapeze Switzerland
	Dr. Mohamed Omar, Idea in Motion Egypt
	Prof. Dr. Slim Abdennadher, German University in Cairo
Submission Date:	29 August, 2019

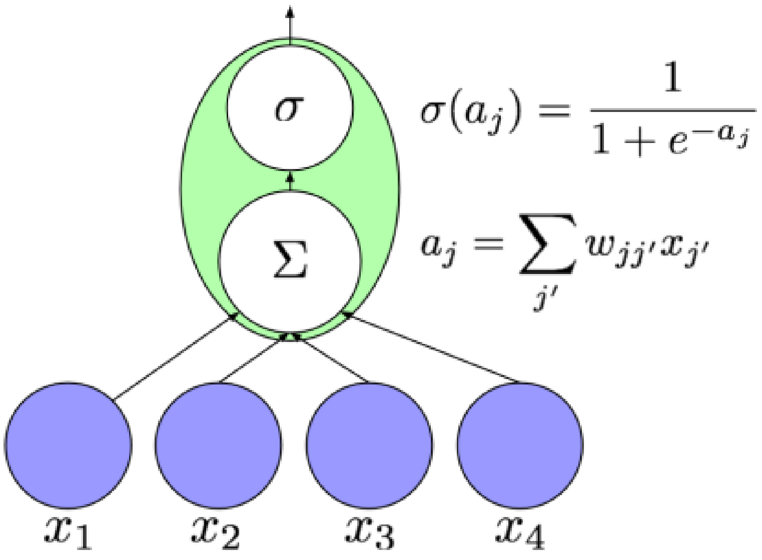

σ
Σ
$\sigma(a_j) = \frac{1}{1+e^{-a_j}}$
$a_j = \sum_{j'} w_{jj'} x_{j'}$
x_1
x_2
x_3
x_4

Figure 2.1: A neuron represented as a circle and the weighted edges as arrows. The activation function is a function of the sum of the weighted edges. [?]

2.2.1 Feed Forward Neural Networks

The most popular form of Feed Forward Neural Network (FNN)s is the Multi Layer Perceptrons (MLP)s [32] [37] [7]. With the absence of cycles, the nodes are arranged into layers, as seen in Figure 2.2, where the layers are classified as an input layer, one or many hidden layers, and an output layer.

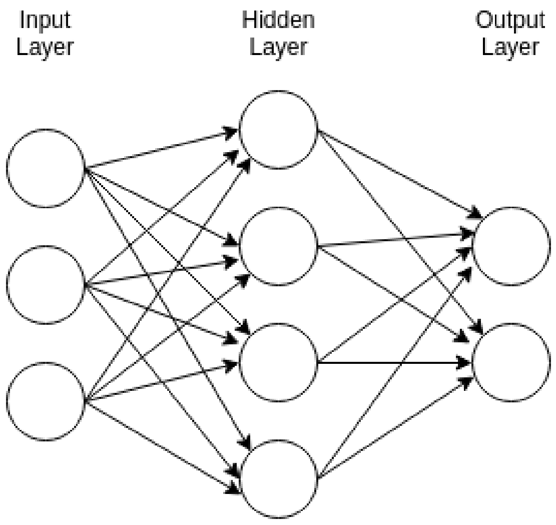


Figure 2.2: A simple Feed Forward Neural Network consisting of an input layer, one hidden layer, and an output layer

The input to FNNs is applied to the input layer. Values of nodes in a given layer, are successively calculated using the values of nodes in the lower layer, until the output is generated at the highest layer: the output layer. This is known as the “Forward pass”. Neural Networks learn by looking at input examples without being explicitly programmed any hard-coded rules about the required task. The learning process is achieved by continuously modifying the weights to minimize an error represented by a loss function $L(\widehat{y}, y)$, which measures the distance between the output y (predicted value of y) and the actual value of y (ground-truth).

The algorithm for training neural networks is back-propagation [32]. Back-propagation uses the chain rule to calculate the derivative of the loss function $L(\widehat{y}, y)$ with respect to each parameter in the network. The parameters (weights) are then adjusted in the direction of less error by an optimization algorithm called gradient descent. This is known as the "Backward Pass"

Sequence Models and the Problem with FNNs

A distinctive feature of the FNNs is the "independence assumption"; that is the presented examples (data points) are assumed to be independent of each other, rendering the FNNs unable to correctly represent input or output sequences with dependencies either in time or space. Examples are words forming sentences, letters forming words, frames of video, snippets of audio clips, DNA sequences, etc. FNNs knows no concept of context when analyzing the given examples, they simply are unable to capture dependencies. With the context being a crucial element when analyzing sequences, a simple solution that addresses this matter is the "time-window" solution, i.e. collect the data from either side of the current input into a window. The fact that the range of useful context (either on the left or the right) vary widely from sequence to sequence and in most cases is unknown makes this approach not very efficient. For example, a model trained using a finite-length context window of length n could never be trained to answer the simple question, "what was the data point seen $n + 1$ time steps ago?"

Another problem with FNNs is that they treat inputs and outputs as "fixed-length vectors". Some representations, such as sentences cannot be represented in such way. Some solutions such as "padding" assume a maximum-length for the inputs and/or outputs. Such approach is not a general one. Thus, it was required to extend these powerful and successful models to better suit the sequential nature of some data, and that is where the Recurrent Neural Networks came into picture.

2.2.2 Recurrent Neural Networks

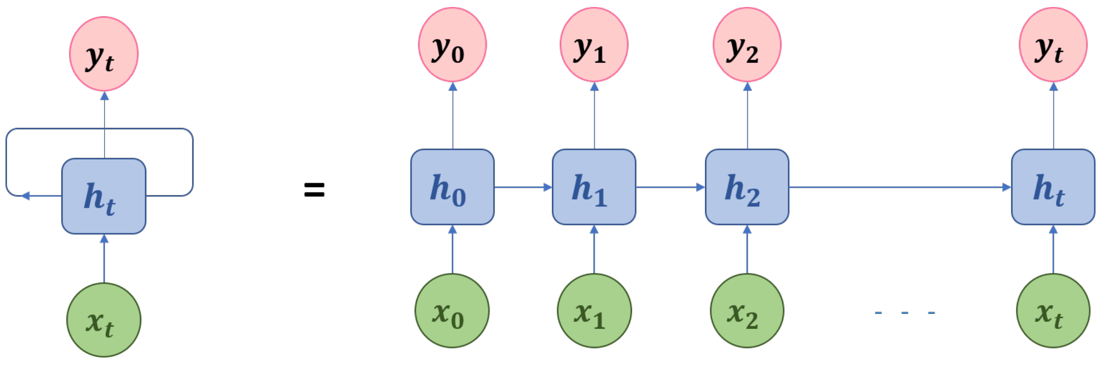


Figure 2.3: A Recurrent Neural Network

ANNs containing cycles are referred to as recursive, or recurrent neural networks. Recurrent Neural Network (RNN)s are models which have the ability to pass information learnt across past data points, while processing sequential data points one by one. Thus they can model inputs and outputs which are correlated either in time or space. They are considered to be neural networks possessing memory.

The RNNs have the following architecture: each hidden layer - commonly referred to as "hidden state" - has two sources of inputs, which are the present data point, and information from the hidden state of the past data point (Figure 2.3). This is how contextual information is propagated across the hidden states of the sequential data points. Equation 2.1 explains how each hidden state nodes values are calculated. Each hidden state $\mathbf{h_t}$ is a function of the present data point $\mathbf{x}$ multiplied by some weight matrix W^x and the previous hidden state $\mathbf{h_{t-1}}$ multiplied by some weight matrix W^h and some bias term $\mathbf{b_h}$. The weight matrices are used to determine how much importance is given

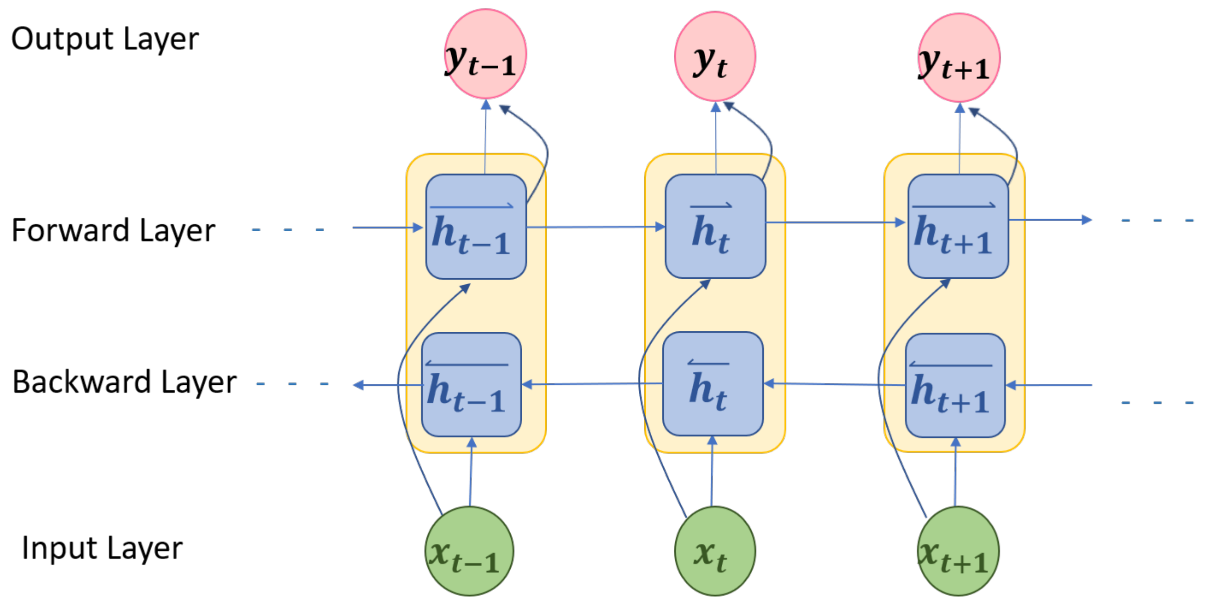


Figure 2.4: A Bidirectional Recurrent Neural Network

Long Short-Term Memory

Long Short-Term Memory (LSTM) [19] is a variation of regular RNNs which was designed as a solution to the vanishing gradients problem. The main idea was to replace the ordinary node with a "memory cell" (see figure 2.5). The cell uses "gates" in order to make decisions about which information to keep and which to discard. It has three gates: output, input, and forget gate, which are analogous to read, write, and reset operations for the memory cell. The gates uses sigmoid as the activation function, where the values of the gates ranges from 0 to 1.

LSTMs are fully described by equations 2.6 , 2.7, 2.8, 2.9, 2.10 and 2.11, where $\mathbf{i_t}$, $\mathbf{f_t}$, $\mathbf{o_t}$, $\tilde{\mathbf{c_t}}$, $\mathbf{c_t}$, $\mathbf{h_t}$ are respectively the input gate, forget gate, output gate, the cell candidate value, the new cell value and the hidden vector.

$$\mathbf{i_t} = \sigma(W^{ih}\ \mathbf{h_{t-1}} + W^{ix}\ \mathbf{x_t} + \mathbf{b_i}) \tag{2.6}$$

$$\mathbf{f_t} = \sigma(W^{fh}\ \mathbf{h_{t-1}} + W^{fx}\ \mathbf{x_t} + \mathbf{b_f}) \tag{2.7}$$

$$\tilde{\mathbf{c_t}} = tanh(W^{ch}\ \mathbf{h_{t-1}} + W^{cx}\ \mathbf{x_t} + \mathbf{b_c}) \tag{2.8}$$

$$\mathbf{c_t} = \mathbf{i_t}\ \tilde{\mathbf{c_t}}\ +\ \mathbf{f_t}\ \mathbf{c_{t-1}} \tag{2.9}$$

$$\mathbf{o_t} = \sigma(W^{oh}\ \mathbf{h_{t-1}} + W^{ox}\ \mathbf{x_t} + \mathbf{b_o}) \tag{2.10}$$

$$\mathbf{h_t} = tanh(\mathbf{c_t})\ \mathbf{o_t} \tag{2.11}$$

LSTMs have many variations such as LSTMs with peehole connections or Gated Recurrent Unit (GRU)s. LSTMs were proven to offer better handling of long range dependencies.

Now that we have examined neural networks, we move forward to discussing the speech recognition problem in the next section.

2.3 Automatic Speech Recognition

The speech recognition problem is defined as follows: given an audio waveform, the task is to find the closest possible transcription to what an accurate human would generate upon listening to that audio. This problem dates back to 1960's, however, the basic Hidden Markov Model (HMM) speech recognition systems originated in mid 1980's. In this section we investigate the mechanics of the ASR systems based on HMMs and then move to the growingly popular "End-to-End" Systems.

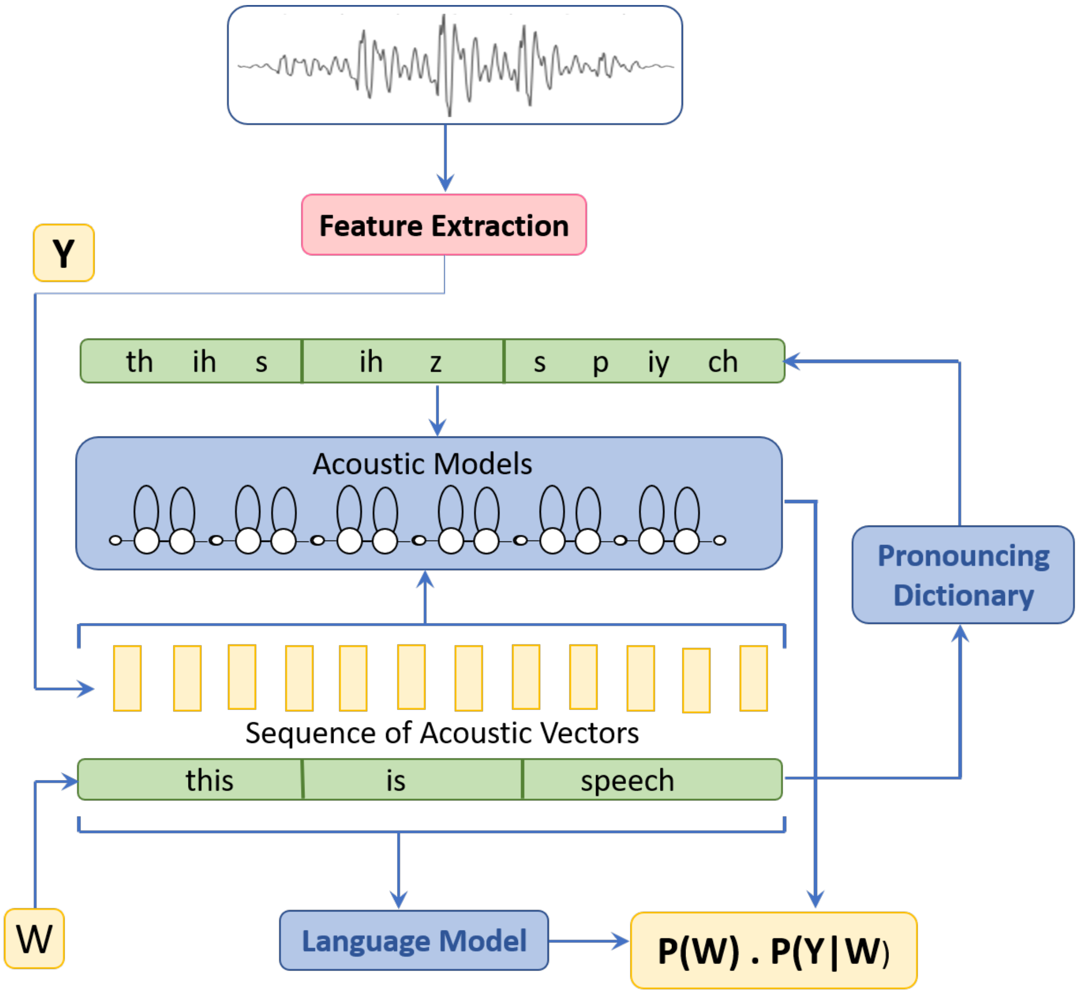


Figure 2.6: An ASR System

Feature Extraction

The premier step in any ASR system is to extract features. That is to turn the raw speech waveforms into a sequence of numerical vectors. The issue, however, lies in the fact that audio signals are constantly changing. For us to be able to deal with them, we make the assumption that for sufficiently small period of time the signals are stationary. Therefore we sample the signal into 25ms frames, with a step of about 10ms so that the frames are overlapping. Then we apply some operations on each frame to extract features. The most widely used feature extraction method is Mel-Frequency Cepstral Coefficents (MFCC), which we explain in brief and illustrate in figure 2.7.

For each frame, the Fast Forier Transform (FFT) is calculated, in order to move from the time domain to the spectral domain. Human ears cannot distinguish between two closely spaced frequencies, specially for higher frequencies; to mimic this effect, we need to know how much energy exists in different frequency regions. This is done by applying special filters called Mel filterbanks. The first filter is the narrowest, and indicates how much energy exists around 0 Hertz. Then the filters become wider as the frequencies get higher because we become less concerned about variations. Afterward, we take the log of the filterbank energies. This is also done to mimic the human ear as we do not hear loudness on a linear scale. Then we compute the Discrete Cosine Transform (DCT) of the log filterbank energies. This is due to the fact that the filterbanks are overlapping, so we compute DCT to reduce the correlation between filter bank amplitudes. The resulting DCT coefficients are referred to as MFCC coefficients. Only 12 coefficients are kept and the rest are dropped, this is proven to improve the ASR performance. These 12 coefficients, together with the normalized energy, they form the feature vector.

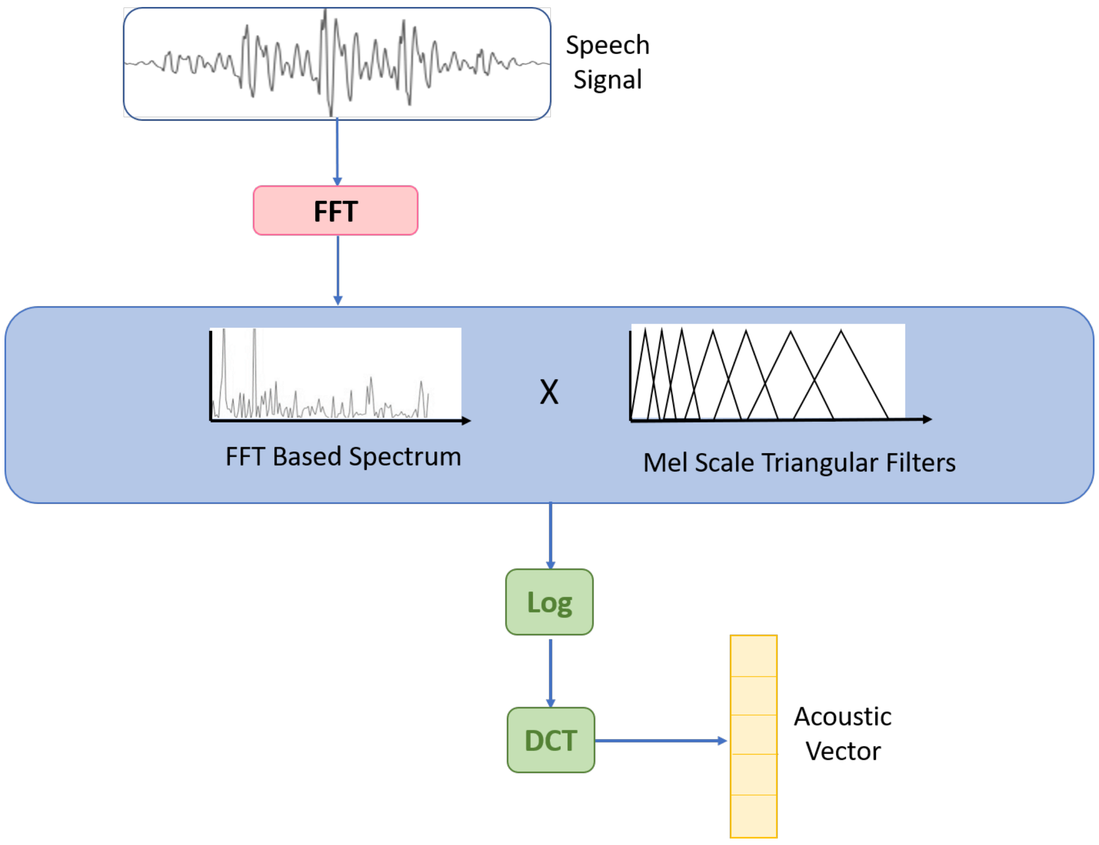


Figure 2.7: MFCC

Acoustic Model

The acoustic model provides a mechanism for computing the probability of a sequence of acoustic vectors Y given a word sequence w. To be able to do this, each word is broken into its constituent phones using the pronouncing dictionary, and we model each phone as a HMM. HMMs are statistical models that are used to predict a sequence of unknown random variables - hence the name "hidden" - from a set of observed variables. Here the unknown variable is the phones sequence, and the observed variables are the sequence of acoustic vectors. HMMs are Finite State Machines (FSM) based on the "Markov Assumption" which states that, in order to predict the future (next state), all we need to know is the present (current state) only, neglecting the past (previous states). *i.e.* $P(q_i|q_1, q_2, ..., q_{i-1}) = P(q_i|q_{i-1})$. Each phone model has an entry state and an exit state which are used to connect different phone models together forming words, and words to be connected together forming sentences. The states are connected by arrows which represent the probabilities of moving from one state to another. Thus, a_{ij} represents the probability of moving from state i to state j. Each time t, the HMM changes state moving from state i to j with probability a_{ij}, and generating an acoustic vector $\mathbf{y_t}$ with probability $b_j(\mathbf{y_t})$. HMMs are also based on the "Output Independence Assumption", which states that the probability of an output observation o_i depends solely on the state q_i that generated that observation, not on any other state or any other output observation. From that, the HMM is fully described by:

1. $Q = q_1, q_2, ...q_n$ A set of n **states**.
2. $A = a_{11}, .., a_{ij}, .., a_{nn}$ A **transition probability matrix A**, where a_{ij} is the probability of moving from state i to state j
3. $O = o_1, o_2, ..., o_T$ A sequence of **T observations**
4. $B = b_i(o_t)$ A sequence of **output probabilities**, where $b_i(o_t)$ is the probability of state i generating observation o_t

2. **Prefix Search Decoding**
   Prefix Search Decoding works by calculating the probabilities of successive extensions of prefixes of label sequences. Despite Prefix Search Decoding being slower, it is guaranteed to identify the most probable label sequence. The drawback here is that the number of prefixes that must be expanded grows exponentially with the length of the input sequence. To overcome this issue, the authors of the paper make use of an observation which is that the outputs of a CTC network form spikes of labels with strongly predicted blanks. Using this observation, a certain threshold is chosen, and points with blank probabilities higher than that threshold are chosen as boundary points, forming sections or regions. For every region, we calculate the most probable label sequence for each region separately, and then we concatenate the results to get the final label sequence.

2.3.3 Deep Speech 2: End-to-End Speech Recognition Model

In this section, we examine a case study for end-to-end ASR models which is Deep Speech 2 [2]. We discuss the model architecture presented in the paper and refrain from describing the datasets used since they are language specific (The authors use their model for both English and Mandarin). We also do not discuss the training procedure or settings since we shall discuss our own settings for training this model on German in the methodology chapter 3.

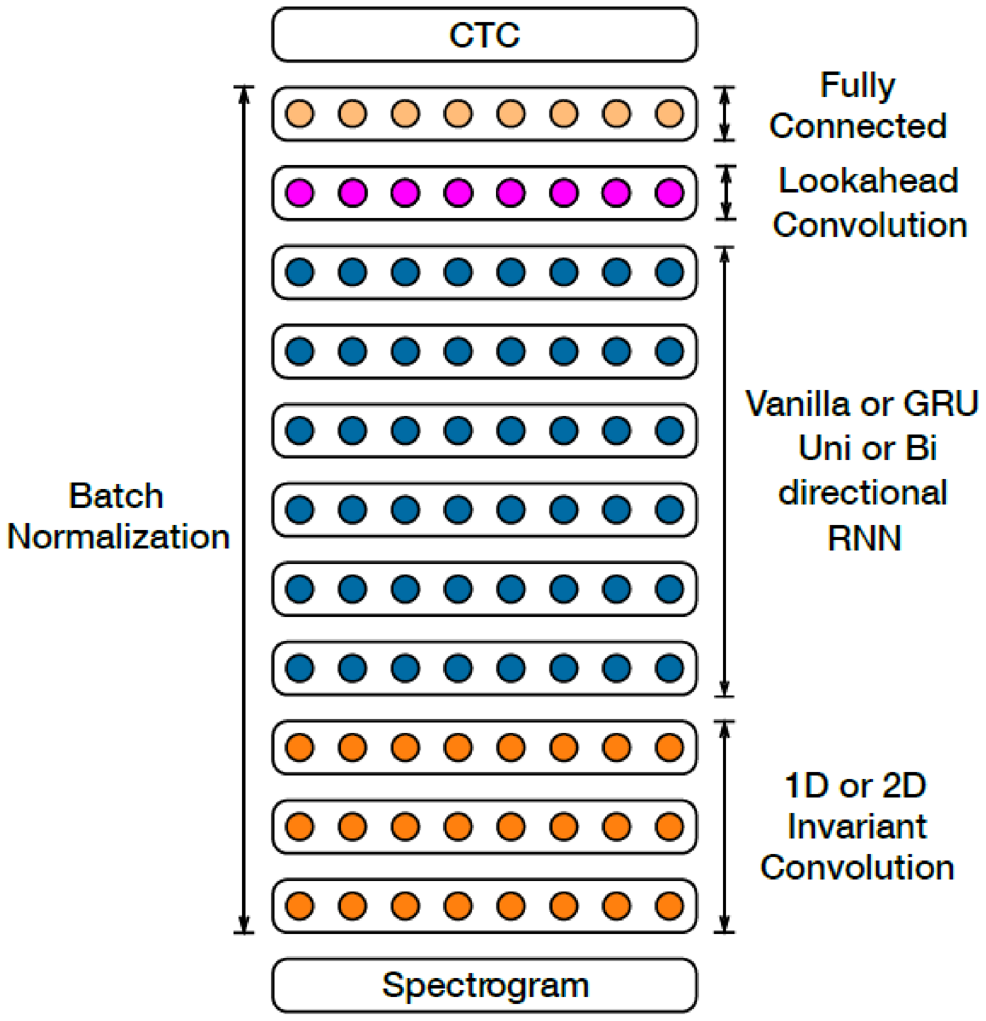

CTC
Fully Connected
Lookahead Convolution
Vanilla or GRU Uni or Bi directional RNN
Batch Normalization
1D or 2D Invariant Convolution
Spectrogram

Figure 2.8: Deep Speech 2 Model Architecture [2]

Deep Speech 2 is an end-to-end ASR model inspired by previous work both in ANNs and speech recognition. It makes use of RNNs, Convolutional Neural Network (CNN)s and CTC. The model

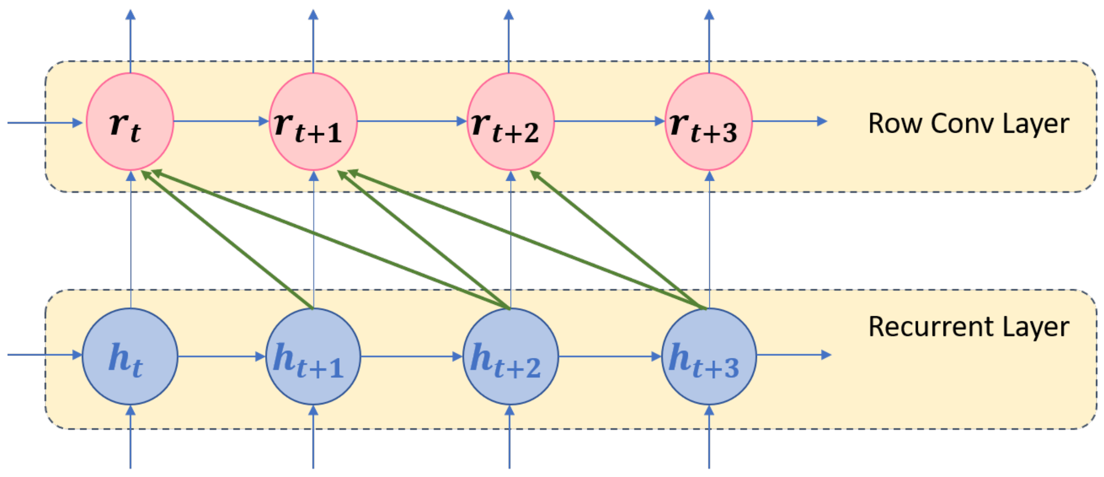


Figure 2.9: Look-ahead Convolution architecture with future context size of 2

"fine-tune" the model using a smaller labeled dataset to a specific task such as text classification, question answering, etc. The features learned from the first task has to be generic in order to be used in solving the second task, in case of text analysis, features are "word embeddings". Word embeddings map words to a high-dimensional vector space, where different words with similar meanings are grouped together in the vector space. Fine-tuning pre-trained language models liberates us from training from scratch which requires large datasets and long time for the model to converge. In section 2.4.4, we shall discuss Bidirectional Encoder from Transformer (BERT) [11], which is a general language model proposed by google that could be pre-trained once, and fine-tuned for many downstream tasks, but prior to that, we discuss in the following three sections the underlying theory of some components in BERT, starting by the "Encoder-Decoder architecture", and going to the "Attention Model" and the "Transformer" which is the core component in BERT.

Encoder-Decoder architecture has been marked broadly useful in modeling sequence-to-sequence models, where the input is a sequence, and the output is as well a sequence. An example is machine translation, where a model is trained to find an output sentence y which maximizes the conditional probability of y given an input sentence x. We take a closer look at it in the next section.

2.4.1 Encoder-Decoder Architecture

The popular encoder-decoder architecture was first proposed by Cho *et al.* (2014a) [9] and Sutskever *et al.* (2014) [34]. This system consists of two RNNs which work together as an encoder-decoder pair (Figure 2.10). The encoder encodes a variable-length sequence (e.g. a sentence) into a fixed-length vector which we call summary vector **c**. The decoder then uses this fixed-length vector to generate a variable-length output sequence. The vector **c** has information from each data point in the input sequence.

The encoder is a RNN which has a hidden state $\mathbf{h_t}$ updated at each time step according to equation 2.17, where f is a non-linear activation function; Sutskever *et al.* (2014) [34] uses LSTMs for this purpose while Cho *et al.* (2014a) [9] uses a variation of LSTMs instead.

$$\mathbf{h_t} = f(x_t, \mathbf{h_{t-1}}) \tag{2.17}$$

The decoder is also a RNN that is trained to generate the output sequence $\mathbf{y}$ one by one. Its hidden state $\mathbf{s_t}$, which is used to generate the output y_t is calculated according to equation 2.18, where it is a function of the previously generated symbol y_{t-1}, the previous hidden state $\mathbf{s_{t-1}}$ and the summary vector **c**. Similarly, the conditional probability of the target symbol is given by 2.19

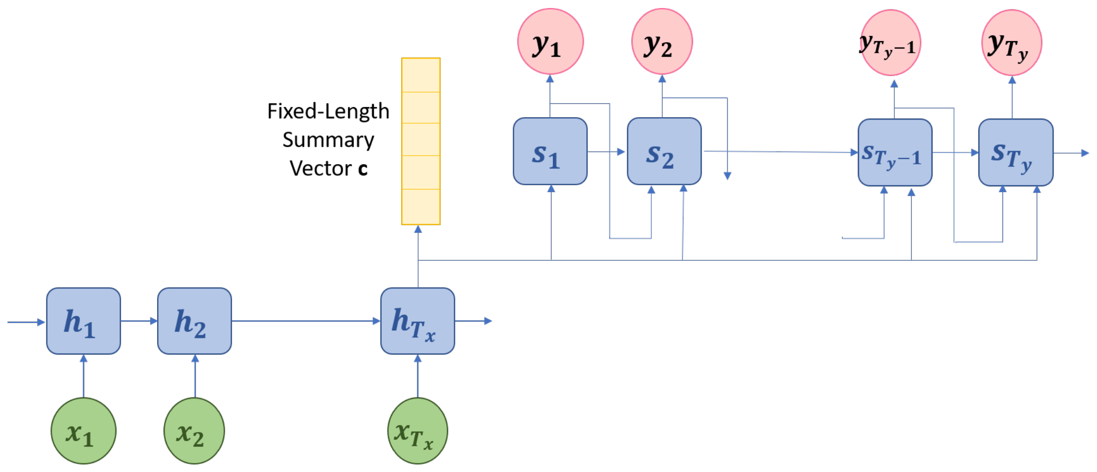


Figure 2.10: An Encoder-Decoder System. The encoder encodes a variable-length sequence into a fixed-length vector **c**. The decoder uses the summary vector **c** along with the previously generated predicted symbol from the previous time step y_{t-1} and the current hidden state $\mathbf{s_t}$ to generate an output y_t

$$\mathbf{s_t} = f(\mathbf{s_{t-1}}, y_{t-1}, \mathbf{c}) \tag{2.18}$$

$$P(y_t|y_1, y_2, .., y_{t-1}, \mathbf{c}) = g(\mathbf{s_t}, y_{t-1}, \mathbf{c}) \tag{2.19}$$

The two components are then trained to maximize the conditional probability of the output sequence $\mathbf{y} = (y_1, y_2, .., y_{T_y})$ given the input sequence $\mathbf{x} = (x_1, x_2, .., x_{T_x})$

The problem with this architecture is that the performance deteriorates as the length of the input sequences increases [8]. This is due to the difficulty of cramming all the necessary information into a fixed-length vector. In order to address this issue, the attention mechanism was introduced by Dzmitry *et al.* [4]

2.4.2 Attention Mechanism

Attention is the ability to focus on important details and discard irrelevant information. Dzmitry *et al.* (2015) [4] proposes modifying the encoder-decoder architecture through arming the decoder with "attention mechanism" which allows it to attend to only parts in the input sentence which are most relevant to the target word in the output sequence. The characteristic feature of this approach is that it doesn't encode all the input sequence into a fixed-length vector as the basic encoder-decoder approach explained in section 2.4.1. Instead, it encodes the input sequence into a number of vectors, and chooses which of these vectors are relevant to the target word in order to make a prediction. This approach performs better on longer input sequences as it is no longer needed to suppress all the information given in the sentence in one fixed-length vector. We demonstrate the model proposed by Dzmitry *et al.* (2015) [4] starting by the encoder, then the decoder.

I. Encoder

No major modifications were performed on the encoder, however, a BRNN was used instead of a uni-directional one (Figure 2.11). This is done in order to gather left and right context as discussed in section 2.2.2. By concatenating the forward hidden state $\overrightarrow{h_j}$ and the backward hidden state $\overleftarrow{h_j}$ for each word x_j in the sequence, an annotation $h_j = [\overrightarrow{h_j}^T, \overleftarrow{h_j}^T]$ for that word is obtained. The decoder would use these annotations later in the attention layer as we will explain.

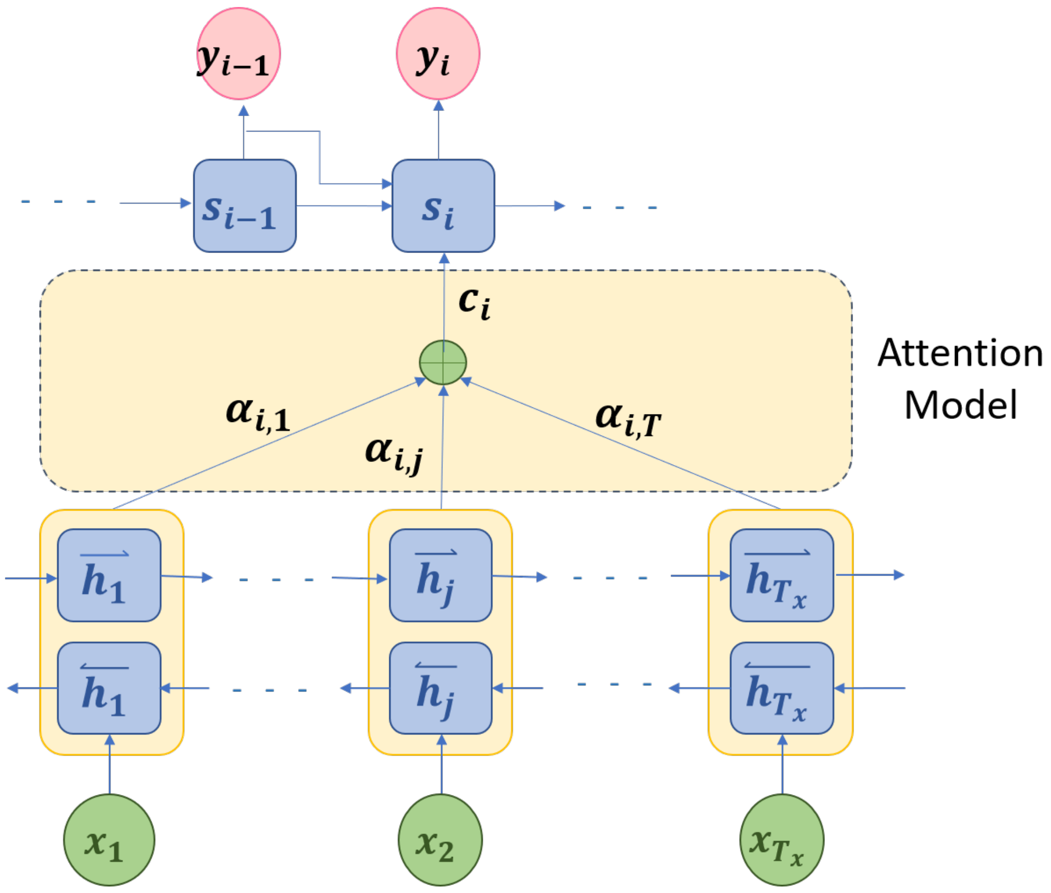


Figure 2.11: The modified encoder-decoder system implementing the attention mechanism. The output y_i has a corresponding context vector $\mathbf{c_i}$ which is a summation of the weighted annotations h_j. The most relevant annotations in the input sentence have the largest weights α_{ij}, while the least relevant annotations have the smallest weights.

II. Decoder

The decoder searches through the input sentence for the most relevant data points when generating an output. It is composed of a RNN that also uses the hidden state $\mathbf{s_i}$ to calculate y_i. The hidden state $\mathbf{s_i}$ of the decoder is calculated according to equation 2.20, where it uses the previously generated symbol y_{i-1}, the previous hidden state $\mathbf{s_{i-1}}$ and a context vector $\mathbf{c_i}$ in predicting the target symbol. The conditional probability of the target symbol is given by 2.21. The major difference here from the basic encoder-decoder approach is that there is a distinct context vector $\mathbf{c_i}$ for each target word y_i (Figure 2.11). The context vector $\mathbf{c_i}$ is calculated as a weighted sum of the sequence of annotations produced by the encoder as seen in equation 2.22. Each annotation h_i carries information from the entire input sequence with strong emphasis on parts closer to the $i-$th point of the input. The weights of the annotations are given by equation 2.23

$$\mathbf{s_i} = f(\mathbf{s_{i-1}}, y_{i-1}, \mathbf{c_i}) \tag{2.20}$$

$$P(y_i|y_1, y_2, .., y_{i-1}, \mathbf{x}) = g(\mathbf{s_i}, y_{i-1}, \mathbf{c_i}) \tag{2.21}$$

$$\mathbf{c_i} = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.22}$$

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{2.23}$$

where $e_{ij} = a(s_{i-1}, h_j)$ is the attention model, or the "alignment model" which resembles the relevance between the target output at position j, and the inputs around position i in the source sentence. The alignment model a is a simple FNN which is trained with the other components.

Using the attention mechanism, there is no need to suppress all information of the input sentence into a single vector, instead the decoder knows for the output y_i where the relevant information lies in the source sentence through the context vector c_i

In 2017, Google came up with a novel architecture using only the attention mechanism and eliminated the use of RNNs, they called their model "The Transformer" [35] and we explain the idea behind it in the next section.

2.4.3 The Transformer

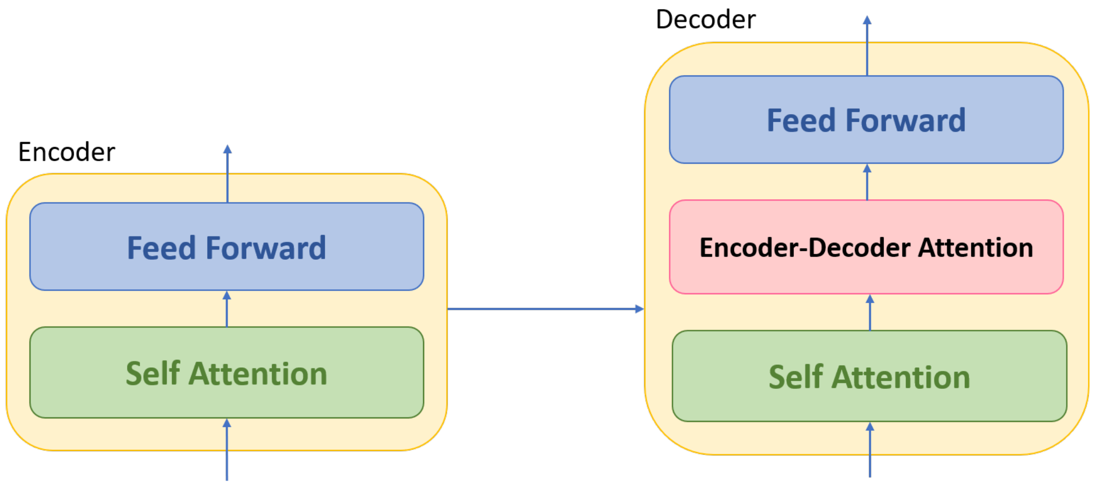


Figure 2.12: The Transformer architecture

Problem with Recurrence

As seen in equation 2.1, RNNs calculate the hidden state as a function of the current data point and the previous hidden state. This sequential nature of calculation largely suits the sequential nature of natural languages, however, this introduces a major problem as it hinders parallelization among the training inputs. This issue is manifested when the input sequences are of longer lengths. The Transformer abandons recurrence in order to achieve more parallelization and efficiency in performance.

The Transformer follows the prominent encoder-decoder architecture, however, introducing major modifications to both the encoder and the decoder. Both of them make use of what is called "Self-Attention" or "Intra-Attention".

I. Encoder

The encoder consists of N layers, each layer made up of two sub-layers. The first sub-layer is called "Self-Attention" sub-layer. This is one of the most influential changes proposed in the Transformer. What self-attention does is that it calculates the relevance of each word in the sequence to every other word in the same sequence. To demonstrate the gain we achieve with self-attention, consider the following example which highlight the problem of linking the pronouns to their antecedents: "The animal did not cross the street because *it* was too tired." A question very simple to humans such as "what does *it* refer to: the animal or the street?" isn't that simple to a model implementing no self-attention. This is owing to the fact that it did not learn any link between "it" and its antecedent when encoding the input sentence. Self attention allows the model to *attend* to other positions in the sequence when processing a certain position, in order to get a better encoding for this word (See Figure 2.13). The second sub-layer is a simple fully-connected feed-forward neural network. A residual connection [15] is applied to each sub-layer, then layer normalization [3] is performed.

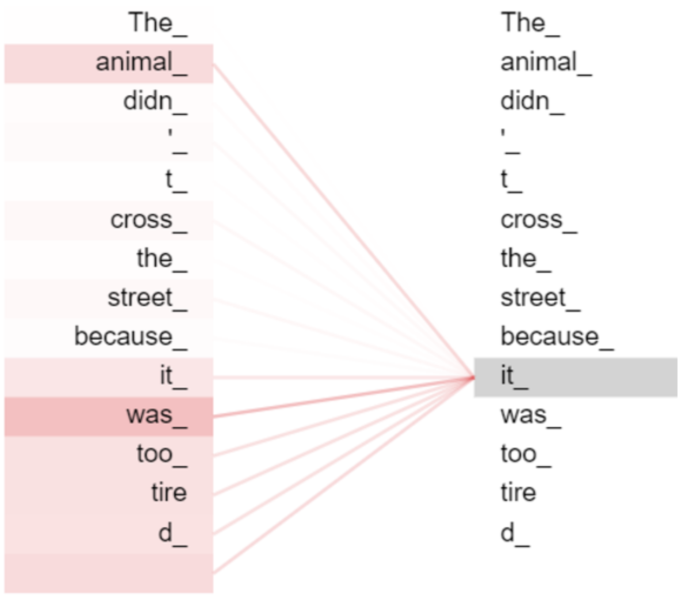


Figure 2.13: Self Attention Weights

II. Decoder

The decoder is made up from N layers as well, with each layer composed of three sub-layers: a self-attention layer similar to the self-attention layer implemented in the encoder, however, modified so that each word can attend only to words earlier in the sequence and not to consequent words, a feed-forward network, and an additional layer which implements the encoder-decoder attention as explained in section 2.4.2. As in the encoder, residual connections are applied on each sub-layer followed by layer normalization.

Because the Transformer has no RNNs or CNNs, the position of each word in the sequence has to be modeled in some way. For this purpose, “positional encodings” [13] [35] are added to the input embeddings at the encoder and the decoder.

The Transformer model utilizes the attention mechanism in three different ways:

1. Self-attention layer in the encoder, where every token in the input sequence attends to every other token in the sequence.
2. Self-attention layer in the decoder, where every token in the output sequence attends to every other token up to that position in the sequence.
3. Encoder-Decoder Attention, where each position in the decoder attends to all positions in the input sequence. This is analogous to the attention implemented in [4]. (Refer to section 2.4.2)

In 2018, Google released a general-purpose “language understanding” model based on the Transformer and called it BERT (Bidirectional Encoder from Transformer) [11]. We demonstrate the idea behind it in the next section.

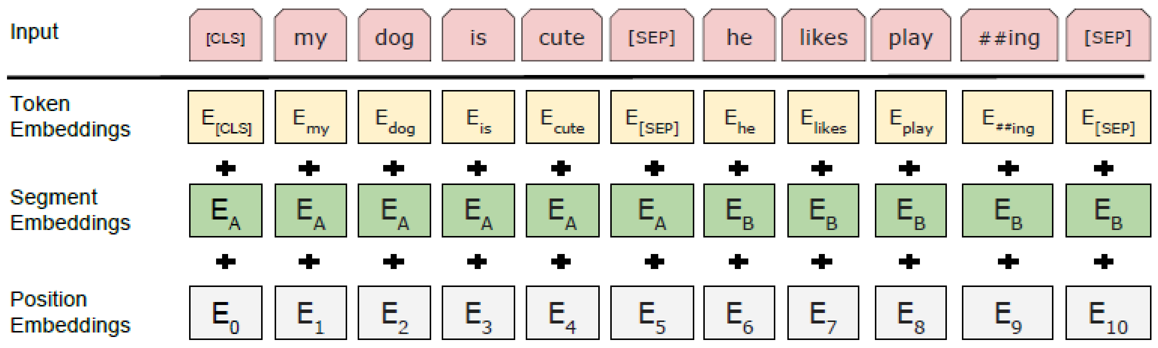

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 2.14: BERT input representation

it too expensive to train, and too much masking percentage means too little context. Because the `[MASK]` token is never present in the fine-tuning phase, the authors employ a technique of replacing the masked word with the `[MASK]` token 80% of the time. 10% of the time it is replaced with a random word, and 10% of the time the word is kept as it is; this is done in order to bias the representation toward the correct word. Since replacing the word with a random one happens only for 1.5% of all input tokens, this approach does not deteriorate the model's performance.

2. **Next Sentence Prediction**
   In order for the model to understand the relationships between sentences, it is pre-trained on a next sentence prediction task as well. When encoding training examples, 50% of the time sentence *B* is the actual next sentence after *A* (*i.e.* given label `IsNextSentence`, and the other 50% of the time, *B* is a random sentence from the training corpus (*i.e.* given label `NotNextSentence`. As an example:

   Sentence A: the man went to the store.
   Sentence B: he bought a gallon of milk.
   Label: `IsNextSentence`

   Sentence A: the man went to the store.
   Sentence B: penguins are flightless.
   Label: `NotNextSentence`

It should be noted that pre-training BERT is an expensive procedure and requires much time and high computational resources. However, since the model is pre-trained on very large text corpus, it enhances the performance on many downstream tasks and the model achieves state of the art results in 11 NLP tasks [11].

Chapter 3

Methodology

In this chapter, the methodology used to implement the system is illustrated. We start with an overview of the proposed system, an interpretation of the choice of architecture and the decisions made is also elaborated. Eventually, the system implementation steps are demonstrated in detail, along with all the methods and procedures.

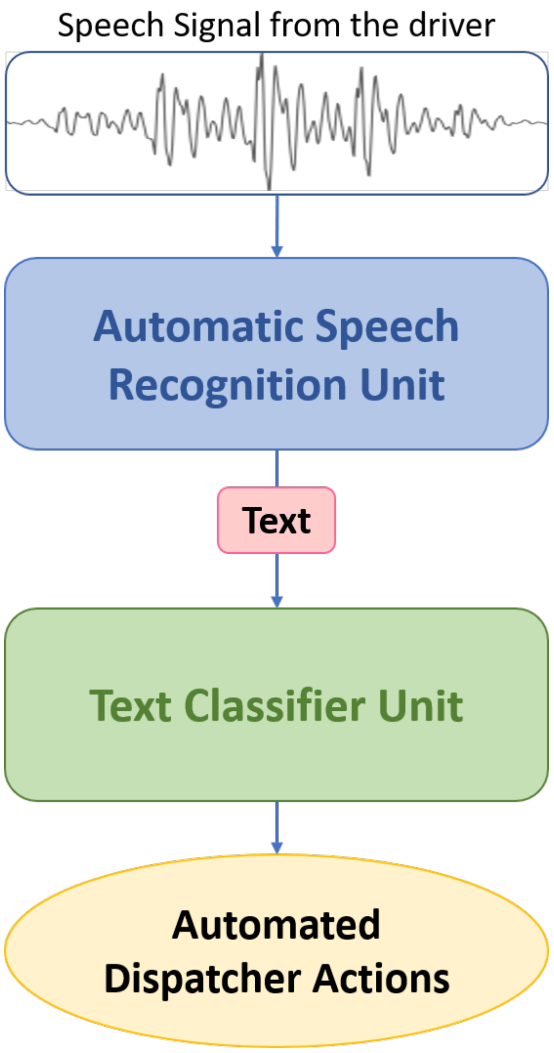

Speech Signal from the driver
Automatic Speech Recognition Unit
Text
Text Classifier Unit
Automated Dispatcher Actions

Figure 3.1: Automated Dispatcher Actions System comprising two sub-systems: Automatic Speech Recognition unit taking input as raw speech from the driver. The text output is fed into a trained Text Classifier which issues the proper actions accordingly.

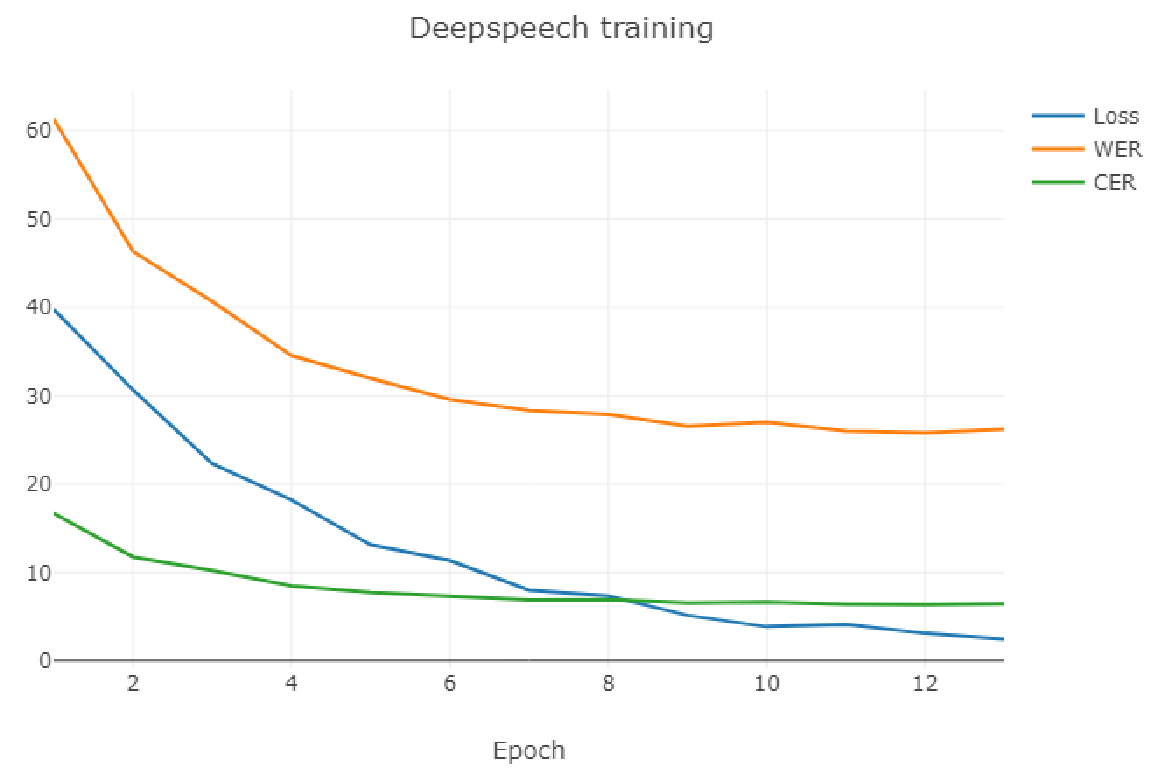


Figure 3.3: Training summary from epoch 1 to epoch 13. The graph shows the WER, CER and loss.

	Train	Dev	Test	Speakers
Tuda-De	160.15h	11.53h	11.9h	179
M-AILAB	233.71h	-	-	5+mixed
Common Voice	274.75h	23.1h	20.34h	4823
Movies	42h	-	-	-
CSS10	16.7	-	-	1
Total	727.31h	34.63h	32.24h	¿5008

Table 3.4: Train, test and dev sets after adding CSS10 and movies data

Training Process and Results

We proceeded with the training using the same configurations after adding all the datasets except SWC. we also added the noise augmentation option available and implemented in the PyTorch model. It applies some changes to the tempo and gain when it's loading the audio. This aims at increasing robustness to environmental noise.

After 19 epochs, the model converged at WER 22.875% and CER 5.629% on the dev set. The training summary is given by table 3.5 and figure 3.4.

The fact that CTC adopts the assumption that the output labels are conditionally independent often makes the outputs suffer from errors that needs linguistic information to be corrected. In attempts to achieve better results, three different improvement techniques were investigated. We discuss them in the following three subsections.

3.4.5 Language Model Decoding

In speech recognition, it is convenient to couple the model with a language model decoding technique in hope for achieving better WER. Since there is more text data available than transcribed audio,

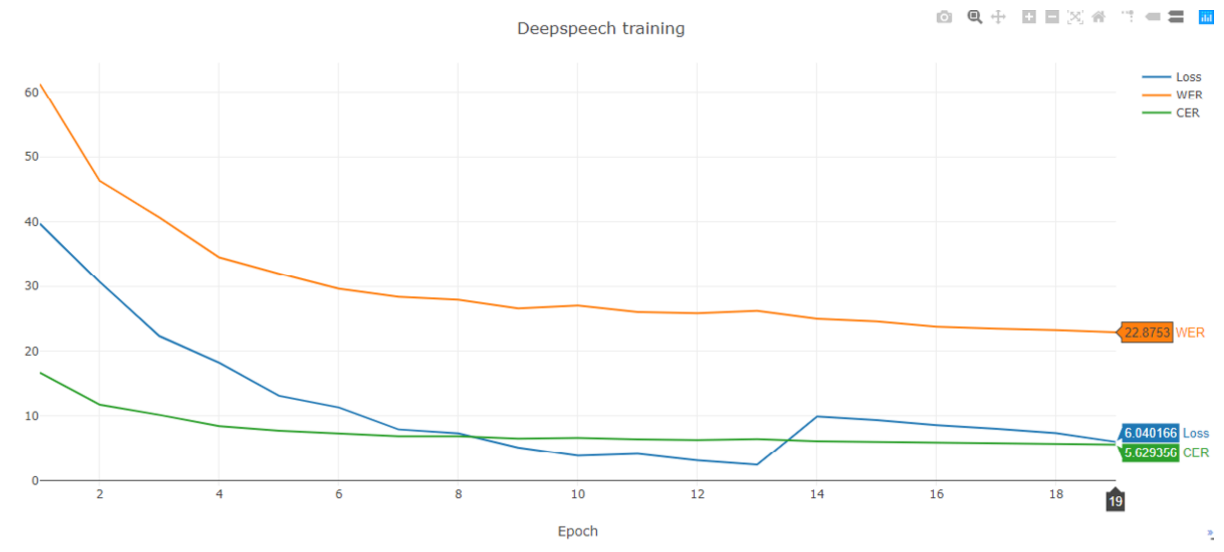


Figure 3.4: Training summary till epoch 19. The graph shows the WER, CER and the loss.

3. **8 Million normalized Mary Sentences**
   We make use of the data used for language modeling by [26], available for download here[15]. The data was in the needed format, with one sentence per line and needed no further processing.

4. **Transcriptions of Speech Recognition Data**
   We also make use of the transcriptions of our collected audio datasets. It was already one sentence per line, cleaned and suitable for use without any further processing.

All of our data was augmented in one text file. We started by installing KenLM dependencies, and then installing KenLM itself. We perform word tokenization on our data using nltk library, then train an n-gram model with Kneser-Ney smoothing using KenLM. The generated output is a `.arpa` file which has a data section with unigram, bigram, ..., n-gram counts followed by the estimated values. Though the `.arpa` format is more readable, the `.binary` format is much faster and more flexible. It remarkably reduces the loading time and our ASR model in fact requires the language model to be in `.binary` format to be used. For these reasons, we binarize the model and obtain the final `.binary` language model file used for decoding. It should be noted that language model decoding with beam search is given by equation 3.1

$$Q(y) = \log(p_{RNN}(y|x)) + \alpha \log(p_{LM}(y)) + \beta wc(y) \tag{3.1}$$

where $wc(y)$ is the number of words in the transcription y. The weight α favors the outputs of the language model, while the weight β encourages more words in the transcription.

We experiment with many data combinations and different values for n which we report in details in the results section, however, our best result was 15.587% WER and 5.806% CER when using all our text corpora, with a 5-gram model and beam-with value of 500. The alpha and beta values were set to 0.9 and 0.2 respectively.

3.4.6 Language Model Re-scoring

It is common as well to arm the model with language model re-scoring technique which performs recognition in two passes. In the first pass, a relatively small language model with a low value of n is used *i.e.* 3 or 4 gram; this often makes the decoding process much simpler. We use N-best method

[15] http://speech.tools/kaldi_tuda_de/German_sentences_8mil_filtered_maryfied.txt.gz

available dataset is "10KGNAD"[29], which is a German topic classification dataset released under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. It includes around 10,000 German news articles collected from an Austrian online newspaper. The articles are categorized into nine topics: Web, Panorama, International, Wirtschaft (Economy), Sport, Inland, Etat, Wissenschaft (Science) and Kultur. The number of articles per class is illustrated in figure 3.5 and the train/test numbers given in table 3.6. We stick to these numbers which are proposed by the author which are 90% articles for training and 10% for testing. No extra pre-processing or cleaning was needed other than lowercasing all the letters and performing the same text cleaning made for the ASR transcriptions and BERT's pre-training data.

3.5.6 Fine-Tuning Process

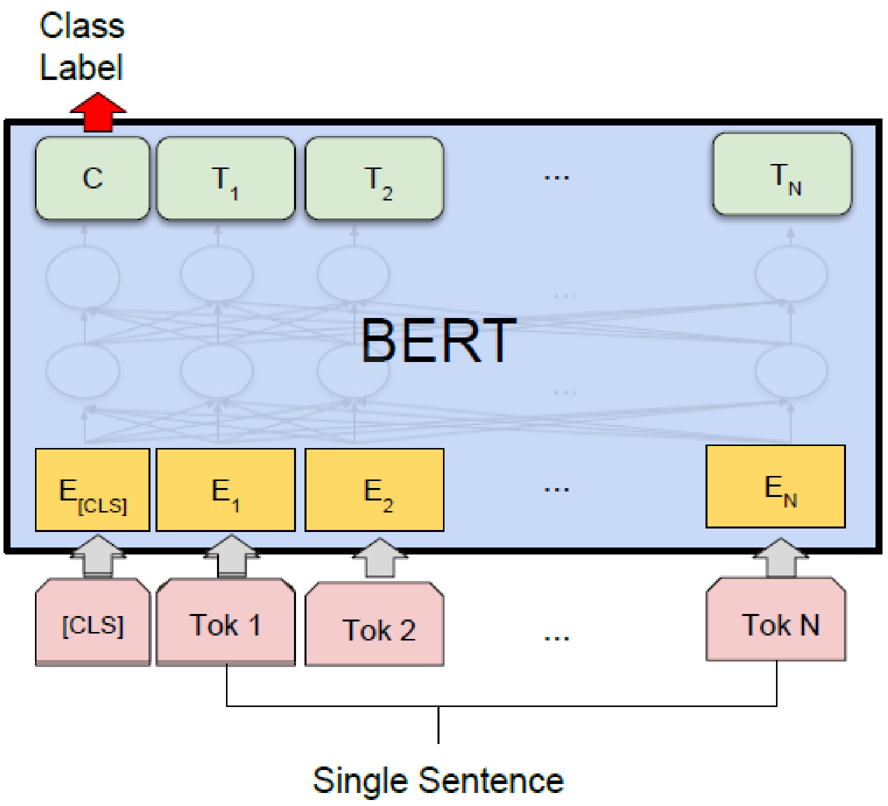
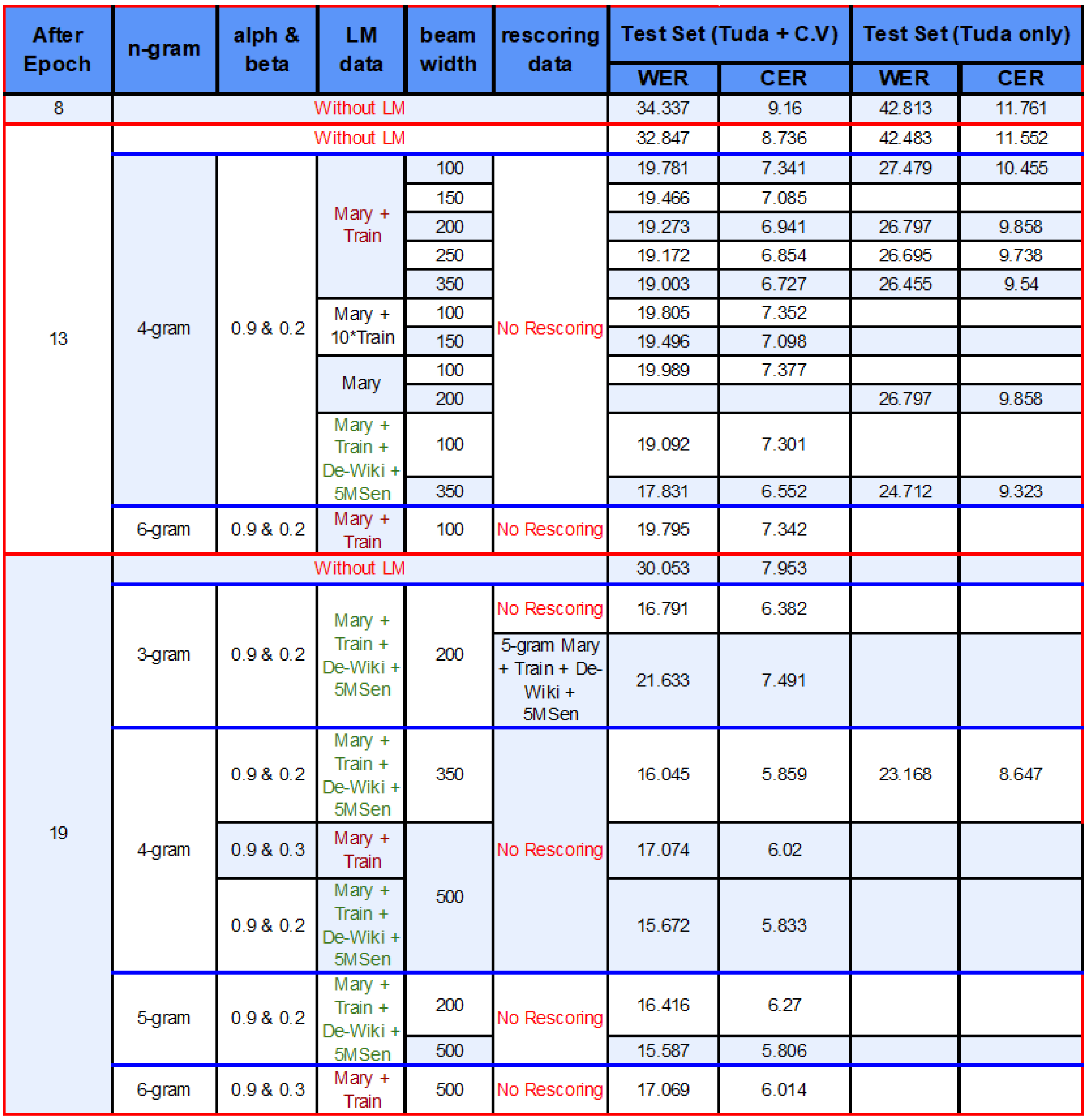


Figure 3.6: The task specific models are formed by building an output layer on top of BERT, so that small number of parameters need to be learned from scratch. The final hidden layer corresponding to the special classification token is used as the classifier output.

Fine-tuning BERT as a classifier is a fairly simple task. Only one additional output layer is added therefore small number of parameters need to be learned from the scratch. The idea is to take the output of the Transformer (*i.e.* final hidden layer) corresponding to the special classification token `[CLS]` as illustrated in figure 3.6. This vector is denoted as $C \in \mathbb{R}^H$ where $H = 768$ is the hidden size. The parameters learned from the scratch during fine-tuning are those for the classification layer $W \in \mathbb{R}^{K \times H}$ where K is the number of labels in our classification problem. For computing the label probabilities, standard softmax function is used. During fine-tuning, BERT's parameters and the classification layer's W parameters are trained jointly to maximize the log-probability of the correct label.

We keep most of the hyper-parameters same as the pre-training, however we change the batch size to 32 and start with an initial learning rate of 2e−5. We tokenize the input data using our vocab

[29]https://tblock.github.io/10kGNAD/



After Epoch	n-gram	alph & beta	LM data	beam width	rescoring data	Test Set (Tuda + C.V)		Test Set (Tuda only)	
						WER	CER	WER	CER
8	Without LM					34.337	9.16	42.813	11.761
13	Without LM					32.847	8.736	42.483	11.552
	4-gram	0.9 & 0.2	Mary + Train	100	No Rescoring	19.781	7.341	27.479	10.455
				150		19.466	7.085		
				200		19.273	6.941	26.797	9.858
				250		19.172	6.854	26.695	9.738
				350		19.003	6.727	26.455	9.54
			Mary + 10*Train	100		19.805	7.352		
				150		19.496	7.098		
			Mary	100		19.989	7.377		
				200				26.797	9.858
			Mary + Train + De-Wiki + 5MSen	100		19.092	7.301		
				350		17.831	6.552	24.712	9.323
	6-gram	0.9 & 0.2	Mary + Train	100	No Rescoring	19.795	7.342		
19	Without LM					30.053	7.953		
	3-gram	0.9 & 0.2	Mary + Train + De-Wiki + 5MSen	200	No Rescoring	16.791	6.382		
					5-gram Mary + Train + De-Wiki + 5MSen	21.633	7.491		
	4-gram	0.9 & 0.2	Mary + Train + De-Wiki + 5MSen	350	No Rescoring	16.045	5.859	23.168	8.647
		0.9 & 0.3	Mary + Train	500		17.074	6.02		
		0.9 & 0.2	Mary + Train + De-Wiki + 5MSen			15.672	5.833		
	5-gram	0.9 & 0.2	Mary + Train + De-Wiki + 5MSen	200	No Rescoring	16.416	6.27		
				500		15.587	5.806		
	6-gram	0.9 & 0.3	Mary + Train	500	No Rescoring	17.069	6.014		

Figure 4.1: ASR Results