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. Kareem Badawi
	Dr. Mohamed Omar
	Dr. Slim Abdenadher
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. Kareem Badawi
	Dr. Mohamed Omar
	Dr. Slim Abdenadher
Submission Date:	29 August, 2019

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

Maggie Ezzat Gamil Gaid
29 August, 2019

Acknowledgments

Abstract

With significant advances in the Artificial Intelligence field, Natural Language Processing (NLP) and contextual analysis techniques have been maturing greatly over the last few decades, with applications rising in many industries. Valuable applications of NLP includes Automatic Speech Recognition systems, Sentiment Analysis and Natural Language Inference.

Although plenty of ink has been spilled on the subject of autonomous vehicles, the practical application of such systems will not take place overnight and there is a long way before we can see self-driving vehicles capable of handling all the driving tasks. Consequently, the present period is regarded as an intermediate stage which aims for more automation in the transportation process as a whole. Such stage will continue up until the switchover to 100% self-driving vehicles is practical and achievable.

The evolution of transport network operators to more automation requires an AI-enabled control center room. In this project, we aim to enhance the capability of transportation systems, enabling them to understand human speech from vehicle driver input and trigger dispatcher actions automatically. As such, an AI dispatcher agent needs to understand a message/call from a driver in order to trigger the necessary action, thus resulting in increased automation inside the control center. For this purpose we present a system which makes use of Automatic Speech Recognition, where the speech input from the driver is transcribed into text and fed into a Text Classifier Unit. The Text Classifier is then trained to issue the dispatcher actions automatically. The resulting is a system with inputs as driver speech, and output as convenient dispatcher actions.

Contents

Chapter 1

Introduction

Transport authorities in the public transport sector are constantly under pressure to offer high quality services, minimize resources and cost, and increase the efficiency of day-to-day operations. From a passenger point of view, it is essential to have punctual connections, real-time information and optimized travel times. This is achieved by having a control center with a human dispatcher regularly communicating with the vehicles drivers through reliable, state-of-the-art communication platforms like Trapeze's on-board systems. The on-board computer can aid the driver in accessing all the necessary information, as well as exchanging data with the control center or requesting a call to the dispatcher.

Current advanced transportation system make use of smart dispatching systems that assists the dispatcher by providing him/her with a work-flow to follow resulting in a more optimized decision-making process.Although such systems supports both dispatchers and drivers and enables them to complete their tasks more efficiently and reliably, the involvement of humans will inevitably introduces errors. As such, we aim to automate the dispatcher side by replacing the human dispatcher with an AI-dispatcher capable of understanding speech from the vehicle driver and triggering actions accordingly. Such approach cut down costs, minimizes effort and eliminates human error.

Natural language processing and contextual analysis techniques are making rapid progress with apparent impact on many applications and industries. Neural Networks also provide state-of-the-art solutions to many rising challenging problems. Speech Recognition field, in particular, is already showing very compelling results and many applications on top of it are successfully and practically deployed.

In light of the above, we define our problem and thesis objective as designing and implementing an automated dispatcher actions system that employs state-of-the-art speech recognition and contextual analysis techniques in order to be capable of understanding speech input from the driver and evaluating situations. It is thus able to issue the optimum decisions and solutions, resulting in more automation, efficiency and lower cost and effort.

This thesis aims at solving the problem by dividing it into two simpler sub-problems: the first problem is modeled as transcribing the input from the vehicle's driver using speech recognition, and the second problem is concerned with analyzing the transcribed information and making decisions accordingly. Thus, our proposed system consists of two sub-systems which form a pipeline:

- the first part of our system is an ASR unit, with input as speech signals from the vehicle's driver. The output is a transcribed form of the passed information.
- the second sub-system is a Text Classifier unit, which takes as input the output text information generated by the ASR unit. It utilizes neural network and state-of-the-art natural language understanding models in analyzing the textual data and issuing a proper action automatically.

Our Thesis outline goes as follows:

- A brief review of Neural Networks: Feed Forward and Recurrent is presented in section 2.2
- In section 2.3, a literature review on Automatic Speech Recognition field including conventional and end-to-end approaches is provided
- Section 2.4 provides the principles for state-of-the-art contextual analysis techniques
- An overview of the proposed system with its sub-systems is given in section 3.1
- In section 3.2, we demonstrate the first sub-system in our pipeline: the Automatic Speech Recognition Unit.
- Section 3.3 sheds light on the second part of our system: the Text Classifier Unit.
- Detailed report and analysis of our achieved results is made available in section 4
- Conclusion, limitations and future work are discusses in section 5

Chapter 2

Literature Review

2.1 Natural Language Processing

Natural Languages refer to languages written and spoken by human beings. They are English, German, French, etc. Humans can easily learn and understand such languages. On the other hand, computers have difficulty understanding these languages because of the ambiguity problem, as computers understand structured and unambiguous programming languages.

Natural Language Processing (NLP) is a field which aims at enhancing the human-computer interaction by giving computers the ability to understand natural languages. It encompasses two sub-fields: Natural Language Generation (NLG) and Natural Language Understanding (NLU). NLG targets making computers generate human-like sentences. NLU focuses on semantics extraction or building a comprehension of the intent. For quite a long time, NLP researches have been striving to build ASR systems and language models for narrowing down the gap between humans and machines.

In this Literature Review, two sub-domains are explored: Speech Recognition and Text Analytics, but at the beginning a quick review of the widely popular Artificial Neural Networks is elaborated in the next section.

2.2 Artificial Neural Networks

Artificial Neural Network (ANN)s are models of computation modeled after the biological neural networks that constitute the human brain. In early researches of Neural Networks, biological resemblance was emphasized [20] [22] [12], however, nowadays it is obvious that ANNs have little in common compared to biological neural networks. That is due to the biological emphasis being abandoned in favor of achieving satisfying computational results.

The basic building block of ANNs are "neurons", commonly called nodes. The set of nodes are connected to each other with weighted edges, with the weights of the edges resembling the strength of the "synapses" between the neurons as suggested by the original biological model. The neurons are represented diagrammatically by drawing them as circles, and the weighted edges as arrows connecting them. Each node has an activation function associated with it, which takes as input a weighted sum of its input nodes (Figure 2.1).

The most popular activation functions are the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, the tanh function $\phi(z) = \frac{e^z - e - z}{e^z + e^{-z}}$, and the Re-Lu function $l(z) = max(0, z)$. The activation function at the output nodes is considered to be task-specific. However, the most popular ones are the softmax function $\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ for $i = 1, .., K$ and $\mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K$, the sigmoid function, or simple linear functions.

There exist many variations of ANNs, the simplest form of them are those whose edges do not form any cycles. These are called Feed Forward Neural Networks.

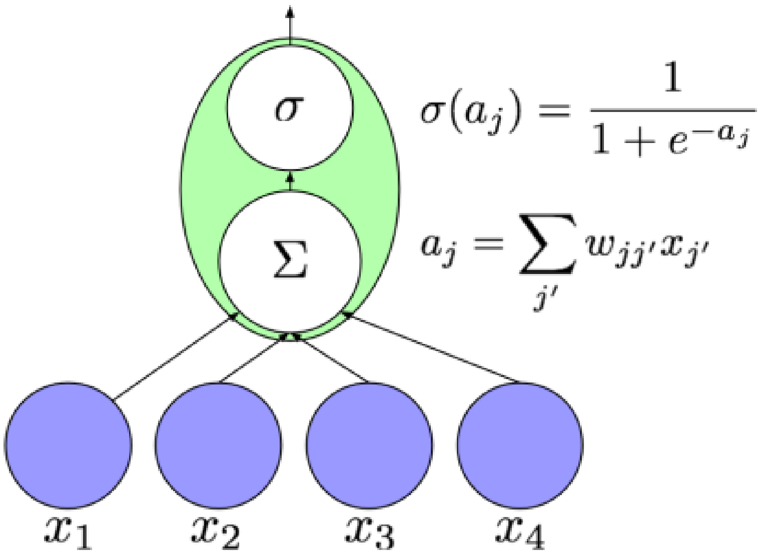

σ
Σ
$\sigma(a_j) = \frac{1}{1+e^{-a_j}}$
$a_j = \sum_{j'} w_{jj'} x_{j'}$
x_1
x_2
x_3
x_4

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 [31] [36] [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.

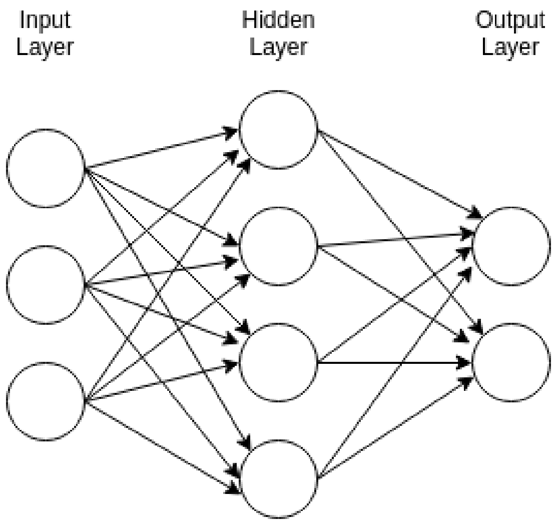


Figure 2.2: A simple Feed Forward Neural Network consisting of an input layer, one hidden layer, and an output layer

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 [31]. 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 that 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 can not 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

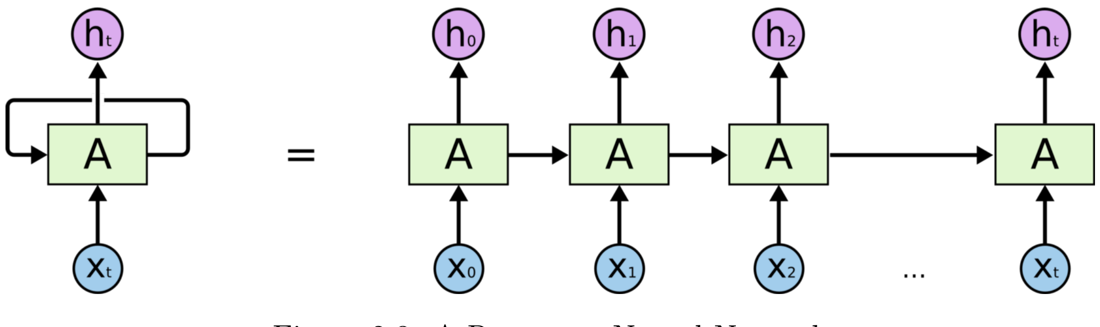


Figure 2.3: A Recurrent Neural Network

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 to both the present data point and the past hidden state. The output $\widehat{\mathbf{y}}_\mathbf{t}$ at time step t is given by equation 2.2, where $\widehat{\mathbf{y}_\mathbf{t}}$ is obtained by applying the softmax function to the hidden state $\mathbf{h_t}$ multiplied by some weight matrix W^y and adding to it some bias term $\mathbf{b_y}$. Similar to the vanilla ANNs, the weights are continuously adjusted to minimize a cost function. This is done using an algorithm called Backpropagation Through Time (BPTT) [37]

$$\mathbf{h_t} = \phi(W^x\mathbf{x} + W^h\mathbf{h_{t-1}} + \mathbf{b_h}) \tag{2.1}$$

$$\widehat{\mathbf{y}_\mathbf{t}} = softmax(W^y\mathbf{h_t} + \mathbf{b_y}) \tag{2.2}$$

Bidirectional RNNs

A slight variation of the RNNs is the Bidirectional Recurrent Neural Network (BRNN)s [32]. The BRNNs have a slight different architecture which allows it to take into consideration information not only from the present and the past input but also from the future input. Note that by past, present, and future, we not only refer to temporal sequences, but also sequences which have a strong emphasis of the order, but bear no explicit notion of time; this is actually the case with natural languages. In BRNNs each hidden layer is duplicated into two layers (Figure 2.4), one layer takes as inputs the present data point and information from the past data point. This is referred to as the "forward direction". The other layer takes as input the present data point and information from the future data point. This is referred to as the "backward direction". The BRNNs are fully described by equations 2.3, 2.4 and 2.5, where $\mathbf{h_t^{<f>}}$ represents the forward direction of the hidden layers and $\mathbf{h_t^{<b>}}$ represents the backward direction of the hidden layers. The predicted value $\widehat{\mathbf{y}_\mathbf{t}}$ is now a function of both the forward and the backward direction. Considering information from both sides of the sequence instead of the left side only adds much power to the network as the context is understood much better. Consider the following example: "She said, 'Teddy bears are on sale.'", "She said, 'Teddy Roosevelt was an amazing president'" On these two examples, considering "Teddy" as the current data point, the left sequence is the same, however, the right sequence is crucial in understanding the context. Thus for an application like named-entity recognition, using BRNNs adds much gain. One drawback about BRNNs is that the entire sequence is needed before any predictions can be made, therefore for real-time systems it is a bit slow as it introduces some delay.

$$\mathbf{h_t^{<f>}} = \phi(W^{xf}\mathbf{x} + W^{hf}\,\mathbf{h_{t-1}^{<f>}} + \mathbf{b_{hf}}) \tag{2.3}$$

$$\mathbf{h_t^{<b>}} = \phi(W^{xb}\mathbf{x} + W^{hb}\,\mathbf{h_{t+1}^{<b>}} + \mathbf{b_{hb}}) \tag{2.4}$$

$$\widehat{\mathbf{y}_\mathbf{t}} = softmax(W^{yf}\,\mathbf{h_t^{<f>}} + W^{yb}\,\mathbf{h_t^{<b>}} + \mathbf{b_y}) \tag{2.5}$$

Problems with RNNs

TODO: vanishing and exploding gradients [17] [18] [6]

Long Short-Term Memory

Long Short-Term Memory (LSTM) [19] is a variation of vanilla 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". 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,

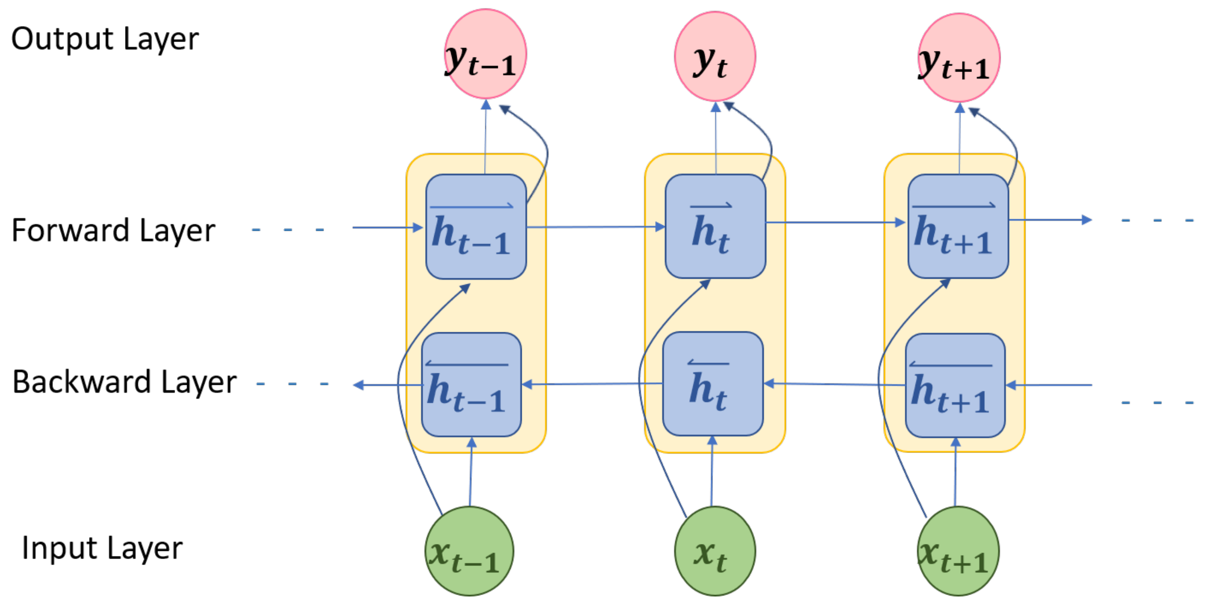


Figure 2.4: A Bidirectional Recurrent Neural Network

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.

TODO 1: continue LSTM

$$\tilde{\mathbf{c}}_\mathbf{t} = tanh(W^{ch}\ \mathbf{h_{t-1}} + W^{cx}\ \mathbf{x_t} + \mathbf{b_c}) \tag{2.6}$$

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

$$\mathbf{i_t} = \sigma(W^{ih}\ \mathbf{h_{t-1}} + W^{ix}\ \mathbf{x_t} + \mathbf{b_i}) \tag{2.8}$$

$$\mathbf{o_t} = \sigma(W^{oh}\ \mathbf{h_{t-1}} + W^{ox}\ \mathbf{x_t} + \mathbf{b_o}) \tag{2.9}$$

$$\mathbf{c_t} = \mathbf{i_t}\ \tilde{\mathbf{c}}_\mathbf{t}\ +\ \mathbf{f_t}\ \mathbf{c_{t-1}} \tag{2.10}$$

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

Problems with LSTM

TODO 2: Problems with LSTMS

2.2.3 Convolution Neural Networks

TODO 3: CNNs

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

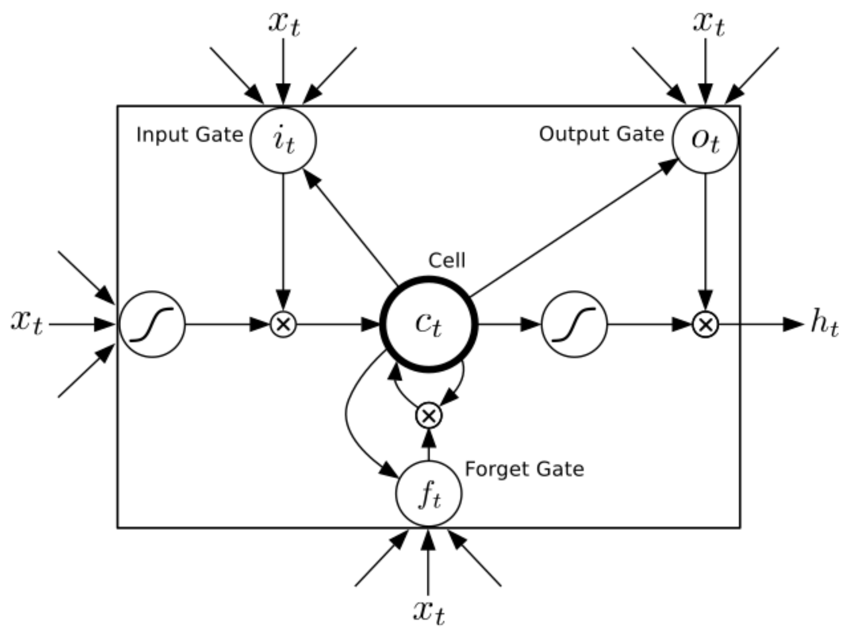


Figure 2.5: An LSTM Cell

2.3 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, moving to the so called "hybrid models" and eventually the growingly popular "End-to-End Systems".

2.3.1 HMM-Based ASR Systems

We begin by demonstrating the main components of an ASR system. As depicted in figure FIGURE, an audio waveform is passed to a "feature extraction" module, which outputs a sequence of acoustic vectors, $\mathbf{Y} = y_1, y_2, ..., y_T$. The audio fragment corresponds to a sequence of words $W = w_1, w_2, ..., w_n$, and it is the objective of the ASR system is to find the most probable word sequence W given a previously unknown audio signal $\mathbf{Y}$. More formally, the target is to find $W = arg\,max_W\ P(W|\mathbf{Y})$. Using Bayes' Rule, this probability can be broken down into two probabilities as shown in equation 2.12.

$$W = arg\,max_W\ P(W|\mathbf{Y}) = arg\,max_W\ \frac{P(W)\ P(\mathbf{Y}|W)}{P(\mathbf{Y})} \tag{2.12}$$

This indicates that we need to find $P(W)$ and $P(\mathbf{Y}|W)$ which maximizes 2.12, in order to find the most probable word sequence W. There are two main components in the ASR system, the "language model" is used to compute $P(W)$, which is the probability of observing the word sequence W independent of the audio signal. The second component, which is the "acoustic model" computes $P(\mathbf{Y}|W)$ which is the probability of the sequence of acoustic vectors Y, given a word sequence W.

Figure FIGURE illustrates the flow of the ASR mechanics. Firstly, $P(W)$ is computed by the language model, then the word sequence W is passed to a "pronouncing dictionary", which breaks the words into "phones". Phones are distinct units of sounds, and they are specific to every language.

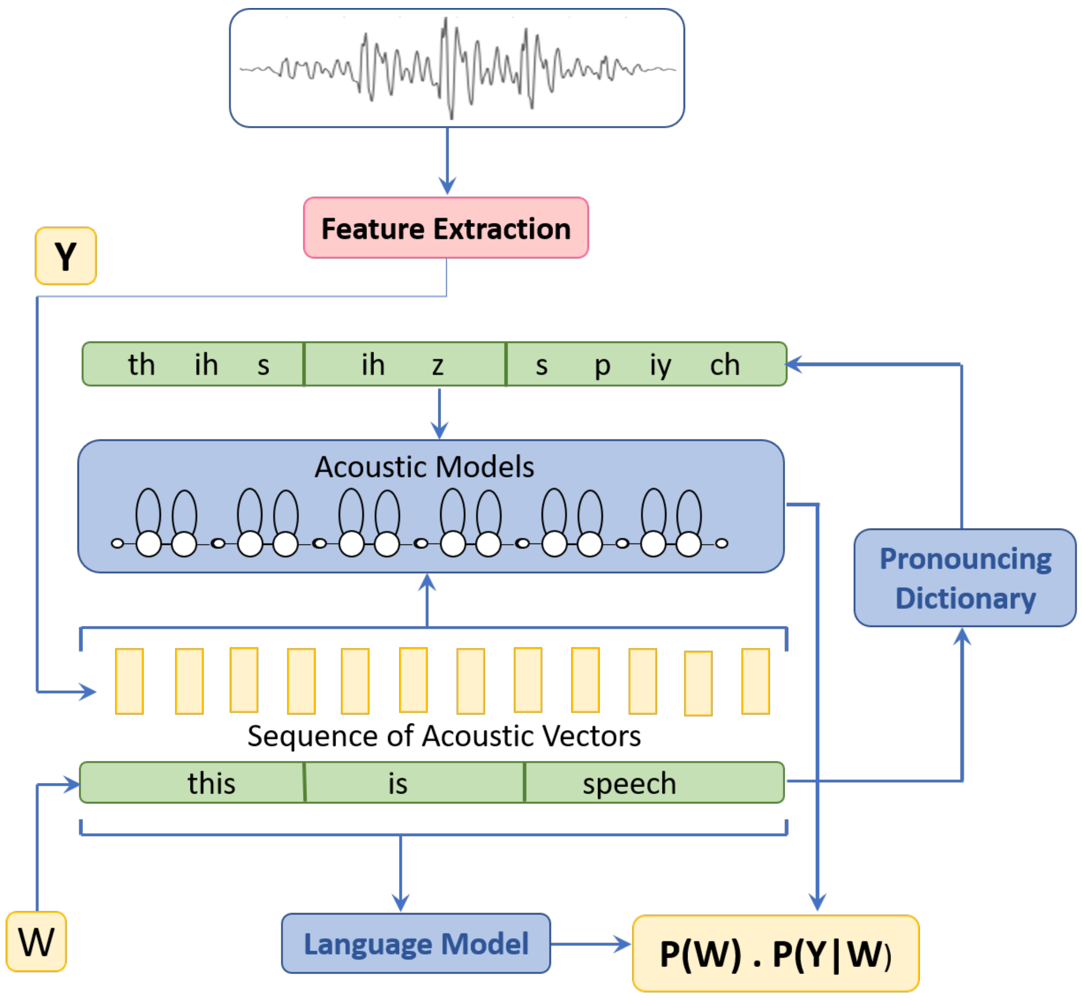


Figure 2.6: An ASR System

For example, English has 44 phones, despite having 26 letters only. That means there 44 different sounds in English. The audio signal is passed to a feature extraction module which outputs numerical features representing the speech signal. For every phone, a statistical HMM model is built, and these models are concatenated together in order to represent the whole sequence of phones making up the utterance using a single model. Then the probability of this model generating the sequence of acoustic vectors Y, given the word sequence W is calculated *i.e.* $P(\mathbf{Y}|W)$. For every possible word sequence W, this process can be repeated until we get the most probable word sequence, however, this is obviously impractical and more efficient methods are used. The process of searching for the most probable word sequence is referred to as "decoding".

We begin by shedding light on the feature extraction module.

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.

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

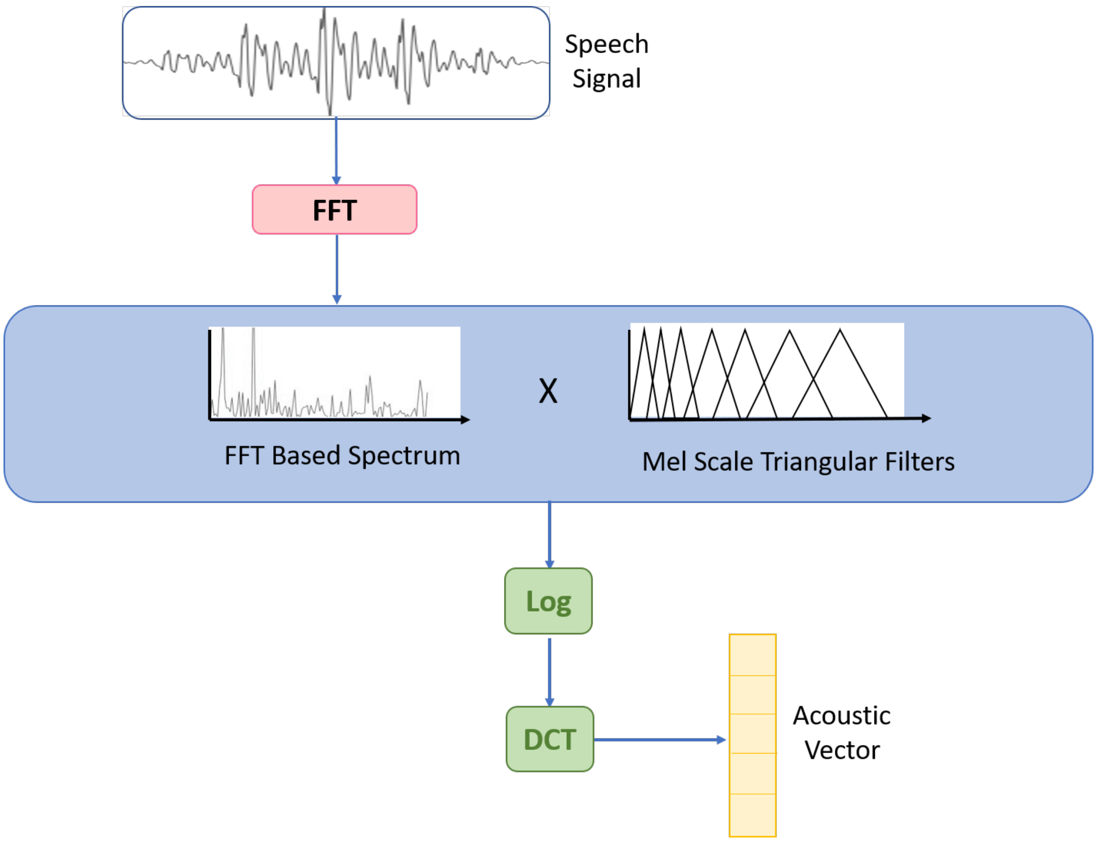


Figure 2.7: MFCC

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.

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
5. $\pi = \pi_1, \pi_2, ..., \pi_n$ An **initial probability distribution** over the states, where π_i is the probability that the HMM will start in state i. [23] [28]

For estimating the parameters A and B, there exists an algorithm called "Baum-Welch Algorithm" or "Forward-Backward Algorithm" for training these parameters. We do not go into the details of this algorithm in this literature, and refer to [23] and [5] for understanding this algorithm.

For decoding, *i.e.* searching for the sequence of words which maximizes equation 2.11, there are two main approaches: "depth first" and "breadth first". In depth first approach, the most reassuring branch is inspected until the end of the speech. In breadth first, which is referred to as "Viterbi Decoding", all hypotheses are inspected in parallel. We elaborate the decoding process in more details; formally, the decoding problem is defined as: given an HMM $M = (A, B)$ and a sequence of observations (acoustic vectors) $\mathbf{Y} = y_1, y_2, ..., y_T$, find the most probable hidden state sequence (phone sequence). The Viterbi algorithm used for decoding is a dynamic programming algorithm that uses the dynamic programming trees or trellis. The algorithm goes by handling the observations sequence one by one, from left to right, and filling out the tree nodes as it proceeds. Each node in the tree has a value $v_t(j)$, this value represents the probability of being in state j after the first t observations and taking the most probable state sequence $q_1, q_2, ..., q_{t-1}$. The probability given by the cell value is described formally as: $v_t(j) = max_{q_1,...,q_{t-1}} P(q_1, ..., q_{t-1}, o_1, ..., o_t, q_t = j|M = (A, B))$. We compute the value of each node by recursively taking the most probable path leading to that node. The value of each cell is computed as given by equation **??**

$$v_t(j) = \max_{i=1}^{N} \ v_{t-1} a_{ij} b_j(o_t) \tag{2.13}$$

where:

$$\begin{aligned} v_t(j) &= \text{previous path probability computed recursively} \\ a_{ij} &= \text{proability of transition from state } i \text{ to state } j \\ b_j(o_t) &= \text{probability of state } j \text{ generating output } o_t \end{aligned}$$

This method is guaranteed to find the best sequence of phones and thus words, however it takes too much time and space. To overcome this problem, pruning of the search space is performed by using "beam search", where every point in time, highest node value is updated, and any node whose value is greater than the highest value plus the "beam width" is ignored. This way, only subset of the most probable probable state sequences are kept in memory at a time. This modification does not guarantee to find the most probable sequence of phones, however, it saves a lot of memory.

Language Model

The sole objective of the language model is to compute the probability of a word w_k given the previous words $W_1^{k-1} = w_1, w_2, ..., w_{k-1}$. "N-grams", are one of the earliest and simplest, but still efficient methods used for language modeling. As implied by the name, n-grams make the assumption that the probability of a certain word w_k, depends only on the previous $n-1$ words. More formally, $P(w_k|W_1^{k-1}) = P(W_{k-n+1}^{k-1})$. N-grams are pretty simple to compute; using frequency counts from textual data and storing them in look-up tables simply gets us the probability distribution. They also capture dependencies and semantics and hence are highly suitable for languages like English due to the fact that word order matters a lot and nearest neighbors contribute largely to the contextual meaning of a word. For more compound languages like German, higher order grams would be a more convenient choice.

2.3.2 Hybrid Systems

TODO 6 : Hybrid Systems PROBLEMS WITH HYBRID MODELS THE NEED TO SEGMENT THE DATA

Although RNNs seemed to be the best suit for sequence models, their use in speech recognition has been limited to hybrid models which do not make use of the full capabilities of the RNNs

2.3.3 End-to-End Systems

In 2006, Graves *et al.* introduced a novel way of using RNNs for labeling unsegmented sequence data which they called Connectionist Temporal Classification (CTC) [14]. We shall demonstrate their paper in this section. The basic idea is to model the RNN outputs as a conditional probability distribution over all possible sequences of labels given a certain input sequence. With that, an objective function is defined that tries to maximize the probability of the correct label sequence. The neural network can then be trained using standard BPTT.

Let L be a finite alphabet of labels and L^* is the set of all sequences over the alphabet L. Likewise, let S be a set of training examples which comprises pairs of sequences $(\mathbf{x}, \mathbf{z})$, with $\mathbf{x} = (x_1, x_2, ..., x_T)$ being an input sequence of real-valued feature vectors, and $\mathbf{z} = (z_1, z_2, ..., z_U) \in L^*$ is a target sequence of labels which is at most the same length as the input sequence x. *i.e.* $U \leq T$. Our goal is to train a classifier h that uses the training examples set S to classify formerly unseen input sequences in a manner that minimizes our "label error rate". Label error rate is defined as the minimum number of insertions, deletions and substitutions required to change the predicted word into the ground-truth word.

The RNN output layer is a softmax layer, consisting of $|L| + 1$ units. The first L units correspond to the probabilities of the L labels of our alphabet, the extra unit correspond to the probability of the output being no label, or "blank". This softmax layer corresponds to the probabilities of the entire possible permutations of labels, hence giving us the probabilities of all possible label sequences for a given input sequence.

For the purpose of investigating the matter in a more formal way, let us define alphabet $L^{'} = \{L \cup blank\}$, and let $L^{'}T$ be all the label sequences of $L^{'}$ of length T, we refer to elements of $L^{'}T$ as "paths" and denote them as π. Also let y_k^t be the probability of the output being label k at time step t. With this, the probability of a certain path, given the input sequence is given by equation 2.14

$$P(\pi|x) = \prod_{t=1}^{T} y_{\pi_t}^t, \; \forall \pi \in L^{'T} \tag{2.14}$$

In equation 2.14, it is assumed that the network outputs at different time steps are independent. This is achieved by not allowing any feedback connection from the outputs to the network itself.

The next step is to remove all blanks and repeated labels from every path. This gives us a new set $L^{\leq T}$ which is the set of all possible label sequences having length less than or equal T defined over the alphabet L without the blank. We then can calculate the probability of a given label sequence l by simply summing the probabilities of all the paths producing that label sequence l. Note that after removing all blanks and repeated labels, many paths would generate the same label sequence. Hence, the probability of a certain label l is given by equation 2.15

$$P(l|x) = \sum P(\pi|x), \ \ \forall\, \pi \ generating \ l \tag{2.15}$$

The output of our classifier should be the label sequence with the highest probability $h(x) = arg\, max_{l \in L^{\leq T}} P(l|x)$.

There are two efficient mechanisms for finding the most probable label sequence:

1. **Best Path Decoding**
   Best Path Decoding is a greedy algorithm based on the assumption that the most probable path, corresponds to the most probable label sequence. The advantage of Best Path Decoding is that it is remarkably easy to compute; the most probable path is the concatenation of the most probable labels for every time step. The disadvantage is that does not ensure finding the most probable label sequence.

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.

2.3.4 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 section 3.

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 itself is a deep neural network consisting of many RNN layers preceded by one or more CNN input layers, and

Sorta Grad Batch Norm Lookahead Convolution

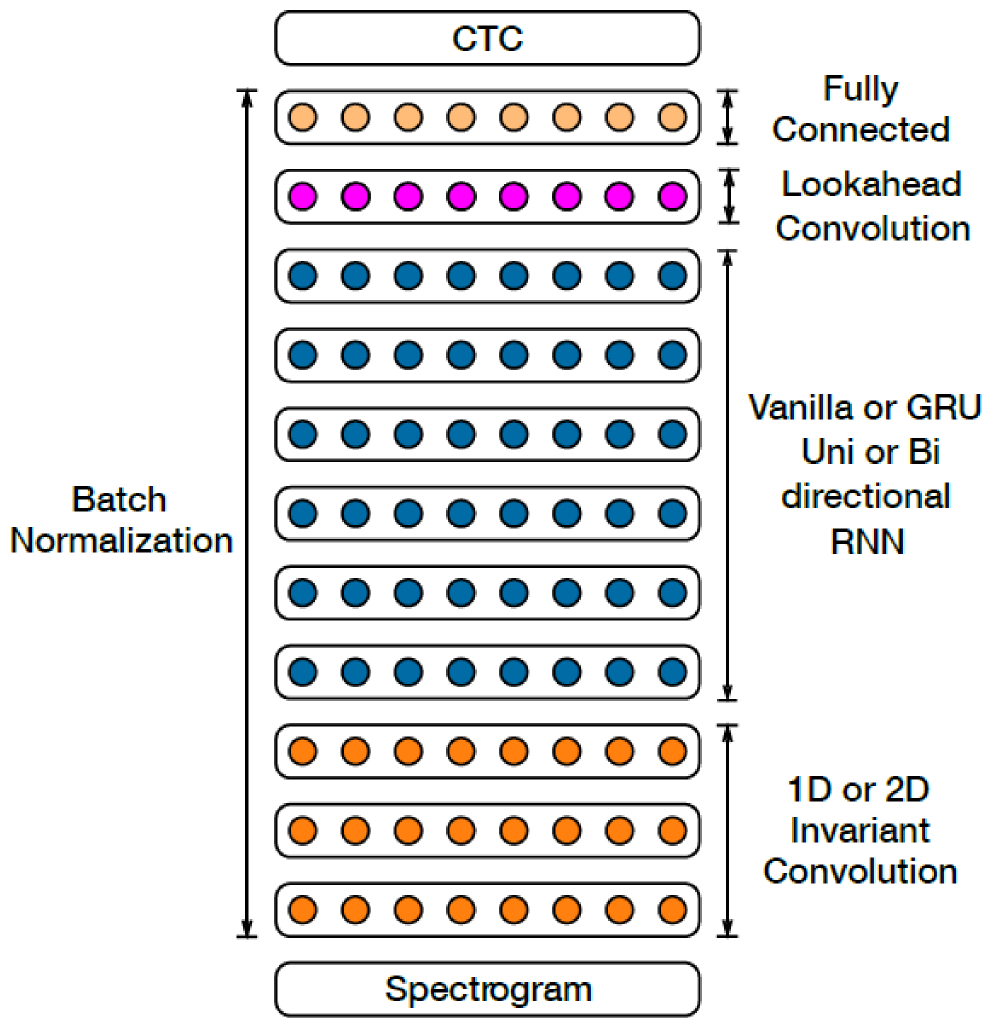

CTC
Fully Connected
Lookahead Convolution
Vanilla or GRU Uni or Bi directional RNN
Batch Normalization
1D or 2D Invariant Convolution
Spectrogram

Figure 2.8: Deep Speech 2 Model Architecture [2]

2.4 Text Analysis

Text analysis is defines as the procedure of extracting information and semantics from written text. Some famous text analysis tasks are text classification, document summarization, named-entity recognition and entity-relation extraction. Text classification is a class of NLP tasks which has been significant to many applications such as spam filtering, language identification, sentiment analysis, and email routing. Transfer learning is the process of using information learned from solving one problem in order to solve another different, but relevant problem. Transfer learning has gained great popularity in many fields like computer vision and text analysis due to the massive amount of data and enormous computational resources required in training deep neural models. Regarding text analysis, the main idea is to "pre-train" a large language model, using large amount of unlabeled text corpora, and then "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) [33]. This system consists of two RNNs which work together as an encoder-decoder pair (Figure 2.9). 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.

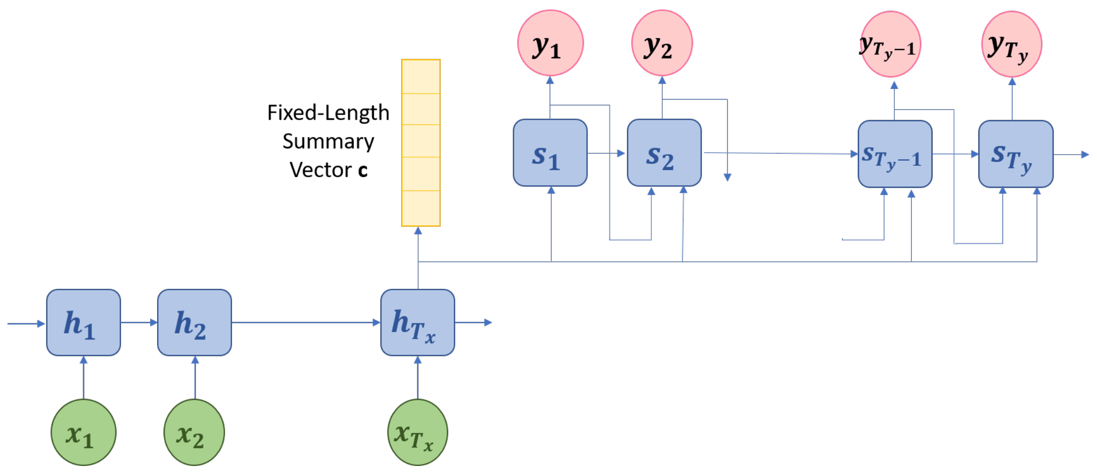


Figure 2.9: An Encoder-Decoder System. The encoder encodes a variable-length sequence into a fixed-length vector **c**. The decoder uses the summary vector **c** along with the previously generated predicted symbol from the previous time step y_{t-1} and the current hidden state $\mathbf{s_t}$ to generate an output y_t

The encoder is a RNN which has a hidden state $\mathbf{h_t}$ updated at each time step according to equation 2.16, where f is a non-linear activation function; Sutskever *et al.* (2014) [33] uses LSTMs for this purpose while Cho *et al.* (2014a) [9] uses a variation of LSTMs instead.

$$\mathbf{h_t} = f(x_t, \mathbf{h_{t-1}}) \tag{2.16}$$

The decoder is also a RNN that is trained to generate the output sequence **y** one by one. Its hidden state $\mathbf{s_t}$, which is used to generate the output y_t is calculated according to equation 2.17, 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 2.18

$$\mathbf{s_t} = f(\mathbf{s_{t-1}}, y_{t-1}, \mathbf{c}) \tag{2.17}$$

$$P(y_t|y_1, y_2, .., y_{t-1}, \mathbf{c}) = g(\mathbf{s_t}, y_{t-1}, \mathbf{c}) \tag{2.18}$$

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 2.3.4. 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.

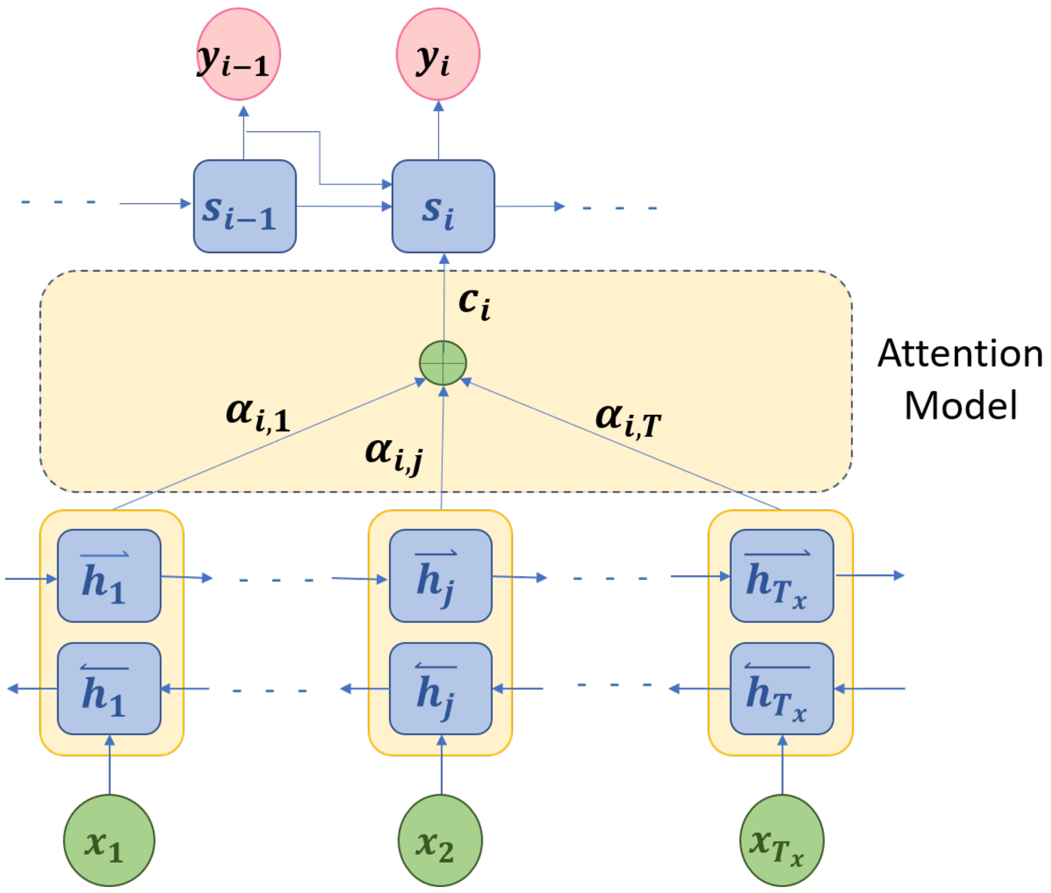


Figure 2.10: The modified encoder-decoder system implementing the attention mechanism. The output y_i has a corresponding context vector $\mathbf{c_i}$ which is a summation of the weighted annotations h_j. The most relevant annotations in the input sentence have the largest weights α_{ij}, while the least relevant annotations have the smallest weights.

I. Encoder

No major modifications were performed on the encoder, however, a BRNN was used instead of a uni-directional one (Figure 2.10). 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.

II. Decoder

The decoder searches through the input sentence for the most for the most relevant data points when generating an on 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 2.19, 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 2.20. 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 2.10). 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 2.21. 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 2.22

$$\mathbf{s_i} = f(\mathbf{s_{i-1}}, y_{i-1}, \mathbf{c_i}) \tag{2.19}$$

$$P(y_i|y_1, y_2, .., y_{i-1}, \mathbf{x}) = g(\mathbf{s_i}, y_{i-1}, \mathbf{c_i}) \tag{2.20}$$

$$\mathbf{c_i} = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.21}$$

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{2.22}$$

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" [34] and we explain the idea behind it in the next section.

2.4.3 The Transformer

Problem with Recurrence

As seen in equation 2.11, 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".

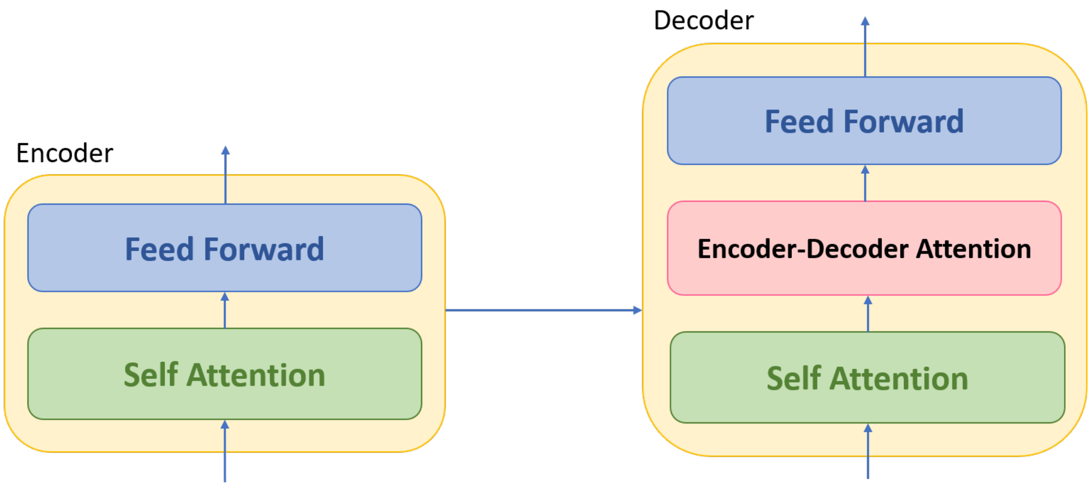


Figure 2.11: The Transformer architecture

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.

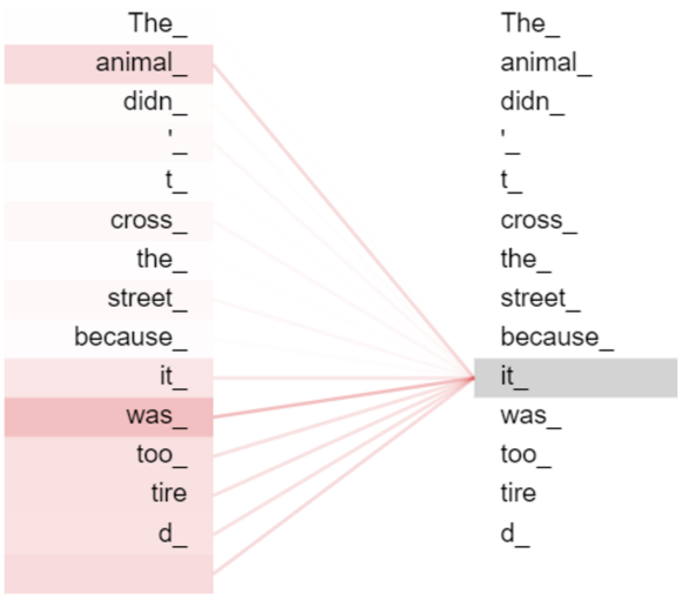


Figure 2.12: Self Attention Weights

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 2.4.1. 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] [34] 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.

2.4.4 Bidirectional Encoder from Transformer (BERT)

As discussed earlier, transfer learning in NLP consists of pre-training a model using a large amount of unlabeled text data on a general task such as language modeling - *i.e.* predicting the next word in a sentence. The next step is to then make use of the learned information in solving a downstream task, such as text classification, text summarization, etc. Previously, transfer learning has been limited to context-free word embeddings. That means that for every word, there is a single word embedding generated, regardless of the context. For example: the word "bank" would have the same word embedding in "bank deposit" and "river bank". Limitations like such have motivated the use of deep neural language models that generate "contextual representations". The idea is to train a neural network model that maps a vector to each word taking into consideration the entire sentence, instead of having only one vector for the word disregarding the context.

BERT [11] is a method of pre-training contextual language representations, which has been inspired by similar works like Elmo [27] and ULMfit [21]. The distinctive feature of BERT over these methods is that in these models, words are conditioned either on the left context only, or conditioned on a trivial concatenation of the left and right context. BERT, in contrast, was described in the paper [11] as "deeply bidirectional", as each word is conditioned on both its left and right context in all layers in a deep neural network. We briefly demonstrate the model architecture and the pre-training tasks.

Model Architecture and Input Representation

BERT is based upon the Transformer [34] described in section 2.2.1, where it consists of a multi-layer bidirectional Transformer encoder. In [11], two versions of BERT were introduced with different sizes;

a relatively small model with 12 layers, 768 hidden size, and a total of 110M parameters, this model was used only for comparison purposes, and the authors refer to it as $BERT_{BASE}$. The larger model has 24 layers, 1024 hidden size, and a total of 340M parameters, they call it $BERT_{LARGE}$ and it achieves state of the art results on many NLP tasks.

As seen in figure 2.13, the input can represents either a sentence pair, or a single sentence. For sentence pairs, the two sentences are separated by a special $[SEP]$ token, and a sentence A embedding is added to the first sentence tokens, while a sentence B embedding is added to the second sentence tokens. For single sentences, only sentence A embedding is used; these are called "segment embeddings". In addition to the segment embeddings, positional embeddings [13] are also added in order to model the sequential nature of text. For every token, its input representation is generated by summing its token embedding, its segment embedding, and its positional embedding. A special $[CLS]$ token is injected at the beginning of each sequence. The output corresponding to this token is used as the output for classification tasks. It should be noted that instead of using whole words as tokens, word pieces [38] are used.

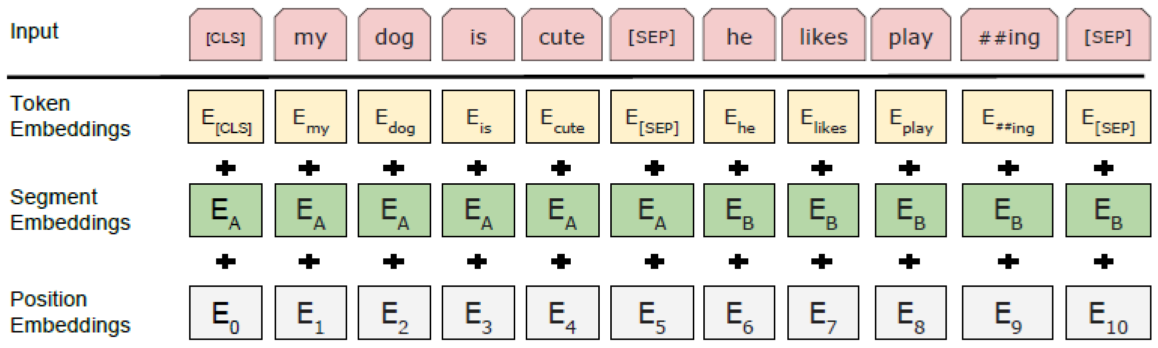

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8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

Figure 2.13: BERT input representation

Pre-training Tasks

1. **Masked LM**
   As discussed earlier, BERT is deeply bidirectional which means it is not trained left-to-right (*i.e.* unidirectional) nor the concatenation of left-to-right and right-to-left (*i.e.* shallowly bidirectional). The major issue about training a deeply bidirectional is that each word would "see itself". To overcome this hurdle, the technique of predicting each word incrementally is abandoned, and another technique called Masked LM (MLM) is used. MLM works by masking out some of the input tokens randomly using a special $[MASK]$ token, and then predicting only those masked tokens instead of predicting the whole input sequence incrementally. The authors of the paper choose to mask 15% of the input sequence as too little masking percentage makes 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 steps and procedures.

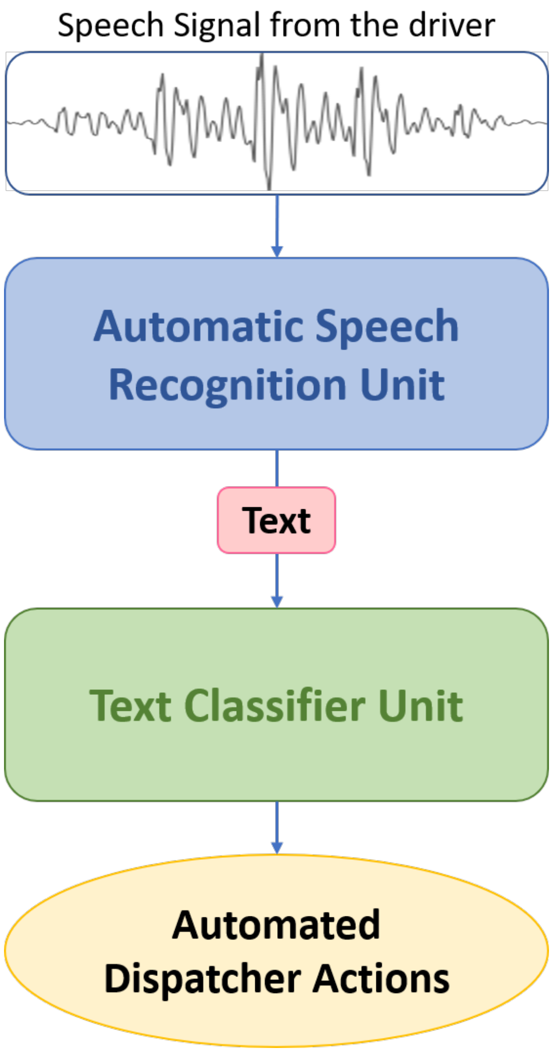

Speech Signal from the driver
Automatic Speech Recognition Unit
Text
Text Classifier Unit
Automated Dispatcher Actions

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 action accordingly.

3.1 System Overview

As discussed earlier, the main purpose of this study is to implement a system able to issue dispatcher actions automatically when provided with audio signals from the vehicle driver, hence, introducing more automation in the control-center.

Our system consists of two sub-systems, as shown in figure 3.1, the first sub-system is the Automatic Speech Recognition (ASR) unit, which takes as input raw speech waveforms produced by the driver, and outputs the driver's information in a text form. The text is then passed to a trained Text Classifier unit, which given the text information from the ASR, issues the corresponding appropriate dispatcher action.

3.1.1 Two sub-systems vs. One speech-based End-to-End System

One might ask why we chose this specific architecture? Why go for two sub-systems and bother with solving the problem of speech recognition? The short answer for this question is simplicity. This approach of dividing the main problem into two problems is much more simpler, since the speech recognition problem is a well defined problem which many researchers tried to approach and solve. The text classification problem is also a well established problem with many state-of-the-art solutions. Another reason for choosing this architecture over an end-to-end classifier (with speech signals as input and dispatcher actions as outputs), is that we do not know what features will be suitable for our problem. The MFCC features are adequate for speech recognition, but there's no proof they will be good for an end-to-end speech-based classifier. The need for a strong analysis of the situation and context as well make it inadequate to implement some keyword spotting system, as it is simply not enough to decide the actions based on some keywords. For the aforementioned reasons, we decide to make one pipeline system constituting two sub-systems, where the output of the first is fed into the second.

3.2 Automatic Speech Recognition Unit

One of the most crucial choices made was the decision about the ASR system since high quality ASR is a chief requirement for speech-based applications. In the next sections, we discuss an interesting German hybrid model based on Kaldi toolkit we compare end-to-end systems to conventional and hybrid ASR systems and demonstrate the model used. We also illustrate the justification for choosing this model.

3.2.1 Hybrid Kaldi-based ASR Model

We started by searching for open-source free German ASRs with satisfying performance. Though many freely available state-of-the-art ASR systems were found, they were mostly English ASRs; this is mainly due to the availability of open-source English training data and the lack of other languages data available for free. Despite being rare, we found an interesting open-source German speech recognition model [25] with freely available training recipes, open source training data, and pre-trained models ready for download and use. This model is based on Kaldi toolkit [10] which is an open-source toolkit for speech recognition research. This model [25] uses Gaussian Mixture Model (GMM) - Hidden Markov Model (HMM) and Time-Delayed Neural Networks (TDNN) [35] [26], following the TED-LIUM corpus recipe example [30] in Kaldi. The model [25] was trained on the German subset of the Spoken Wikipedia Corpus (SWC) dataset[1] (285 hours), Tuda-De dataset[2]

[1]https://en.wikipedia.org/wiki/Wikipedia:WikiProject_Spoken_Wikipedia

[2]http://speech.tools/kaldi_tuda_de/german-speechdata-package-v2.tar.gz

[29] (160 hours) and M-AILABS dataset[3] (237 hours). For now, we only indicate the number of hours for each dataset, however, we shall discuss these datasets in details in section SECTION. The model therefore was trained on a total of 630 hours, and achieved 14.78% Word Error Rate (WER) on the dev set of Tuda-De (11.5 hours) and 15.87% on the test set of it (11.9 hours).

3.2.2 Why End-to-End ASR?

The above model [25] is referred to as "Hybrid Model", making use of both HMMs and neural networks. We found no open-source end-to-end German ASR, and despite the discussed model [25] having satisfying performance, the idea of an end-to-end German ASR was more attractive. This is because end-to-end systems have a much simpler training procedure, as many hand-engineered components and sophisticated pipelines are replaced by neural networks. End-to-end speech recognition is a fast-paced developing area of study with results improving continually. The end-to-end vision liberates us from domain-specific knowledge required for alignment, HMM design, etc. And since end-to-end ASR systems require little task specific components, an end-to-end English ASR can be easily adapted to another language. A further advantage for end-to-end ASRs is that they are more robust to noise and variation in speech and requires no special handling for them. In light of the great advantages for end-to-end systems, we were determined to search for a suitable end-to-end ASR that could be adapted to German without much complexity, and to use the Kaldi based model [25] for comparative purposes only.

3.2.3 Deep Speech 2: Adapting to German

One seemingly convenient model was Deep Speech 2 [2], demonstrated in section 2.3.4. The authors of the paper used their model for both English and Mandarin, two widely different languages. We took that as a proof of concept that the model could be easily adapted with very little changes to other languages including German. We started by mining for open source freely available German speech datasets that could be used since end-to-end ASR systems require large amounts of transcribed audio data. We list the datasets found, their description, problems encountered while dealing with them and the cleaning procedures.

3.2.4 Speech Recognition Data

I. Common Voice

Common Voice is the largest open source, multi-language dataset of voices available for use. It is managed by Mozilla and was collected by volunteers on-line who were either recording samples of audio or validating other samples. Mozilla began work on this project in 2017 and contribution to the dataset continues up till now. Since most of the data used by large companies isn't available freely to researchers, Mozilla's aim was to create a free public dataset to make speech recognition open and accessible to everyone. For our ASR, the German subset of the dataset was selected. It incorporates 340 total hours, with 325 validated hours and 15 invalidated hours which were excluded. The dataset has 5007 speakers but as we discarded the invalidated utterances we end up with only 4823 speakers. All the utterances were in `mp3` format and sampled using a sampling rate of $44KHz$. We converted the audio files to `wav` format and performed down-sampling to obtain sample rate of $16kHz$. We stick to the `wav` format and the $16kHz$ sampling rate for all the used data. We checked for any corrupted files but there were none.

[3] https://www.caito.de/2019/01/the-m-ailabs-speech-dataset/

II. M-AILABS Speech Dataset

M-AILABS Speech Dataset is an open-source multi-lingual dataset provided by Munich Artificial Intelligence Laboratories GmbH. Most of the data is based on LibriVox [4] and Project Gutenberg [5]. We make use of the German subset[6] which is 237 hours 22 minutes with a total of 5 speakers. The data is available in `wav` format and sample rate of $16kHz$ so we perform no extra pre-processing. We also check for corrupted files but all of them were found to be healthy.

III. German Speech Data (Tuda-De)

This open-source corpus [29] is provided by Technische Universität Darmstadt. It has 36 hours read by 180 speakers, and recorded using 5 different microphones simultaneously. They made use of the KisRecord [7] toolkit, which allows for recording with multiple microphones concurrently. Their target was distant speech recognition, thus a distance of one meter between speakers and microphones was chosen. The sentences which volunteers were provided to read were extracted randomly from three text resources: German Wikipedia, German section of the European Parliament transcriptions, short commands for command-and-control settings. The data was in the required `wav` format and sampling rate of $16KHz$, however, there were some corrupted files in the dataset which we discarded. The utterances were divided into 3 sets: train, dev and test and speakers who participated in recording one set did not participate in another. The train set has ≈ 160 hours, dev and test include 11.5 and 11.9 hours respectively.

IV. CSS10: Single Speaker Data

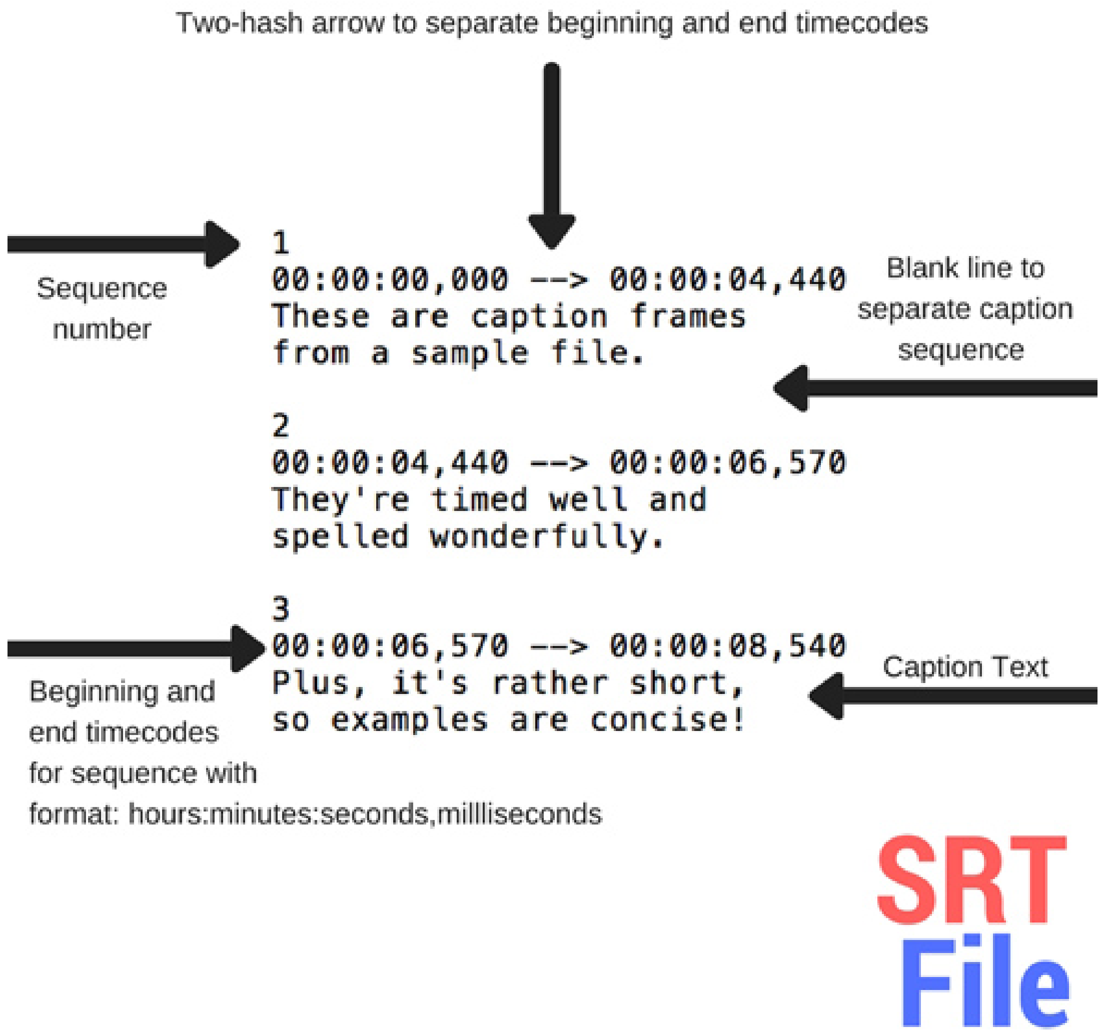
CSS10 is a collection of single speaker speech datasets for 10 languages: Chinese, Dutch, French, Finnish, German, Greek, Hungarian, Japanese, Russian, and Spanish. Each language data consists of audio books from Librivox recorded by a single volunteer. We make use of the German subset of the dataset which is almost 16 hours. The audio files are in `wav` format but sampled at $22KHz$, therefore we down sample to $16KHz$. All of the data was checked for corrupted files, however, none were found.

V. Movies Data

Since end-to-end ASR systems require many hours of transcribed audio, we try to make use of the German movies and series available with their subtitles. Subtitles are the transcribed scripts of the movies/series and are often available in `srt` files. `srt`, which stands for "SubRip Subtitle" file, is a plain text file that contains the subtitles along with their accurate start and end times. Figure 3.2 illustrates the format of `srt` files: each transcription has an ID and also a start and end time separated with two hash arrows (`-->`). The start and end times are used to segment the long movies audio files into utterances of several seconds audio clips.

We make use of 19 German movies and series, with a total of 88 hours, 52 minutes. After pre-processing and extracting only speech audio files (*i.e.* remove all comments like "phone rings", "door opens", "music", etc.), we obtained $44,181$ utterances and a total of 42 hours of German speech data. The last step we perform is converting from `mp3` to `wav` format.

[4] https://librivox.org
[5] https://www.gutenberg.org
[6] http://speech.tools/kaldi_tuda_de/m-ailabs.bayern.de_DE.tgz
[7] http://kisrecord.sourceforge.net


Two-hash arrow to separate beginning and end timecodes
Sequence number
Blank line to separate caption sequence
Caption Text
Beginning and end timecodes for sequence with format: hours:minutes:seconds,millliseconds
SRT File

```
1
00:00:00,000 --> 00:00:04,440
These are caption frames
from a sample file.

2
00:00:04,440 --> 00:00:06,570
They're timed well and
spelled wonderfully.

3
00:00:06,570 --> 00:00:08,540
Plus, it's rather short,
so examples are concise!
```

Figure 3.2: Format of `srt` Files

V. Spoken Wikipedia Corpus

The SWC is an ongoing project where volunteer readers submit Wikipedia articles recorded with their voice. The project aimed at helping users who are unable to use the written version of the articles. These audio resources were time aligned resulting in hundreds of hours of aligned audio data. The SWC incorporates diverse set of topics by large number of volunteers. It is available in English, German and Dutch. It is licensed under a free license (CC BY-SA 4.0). Table 3.1 is a summary of the SWC dataset.

	German	English	Dutch
#articles	1010	1314	3073
#speakers	339	395	145
total audio	386h	395h	224h
aligned words	249h	182h	79h
phonetically aligned	129h	77h	-

Table 3.1: Data used in the first 13 Epochs

Transcriptions Cleaning

We limit the labels to be predicted by our ASR to only lowercase letters from “a” to “z”, apostrophe, space, and the German umlauts ä, ö, ü. We do not add ß as it can be compensated with “ss”. As a result, we perform some pre-processing and cleaning for all our transcriptions:

- All the characters are lower cased.
- Punctuation marks are stripped away.
- Numbers are replaced with their written form, since no digits exist in our labels.
- Foreign characters are discarded.
- $, m^3, km^2, m^2 and are replaced with “dollar”, “kubikmeter”, “quadratkilometer”, “quadratmeter” and “pfund” respectively

After collecting the data, came the step of finding an implementation for the model. In the next two sections, we spot the light on two implementations for the model that we experimented with.

3.3 Deep Speech 2: TensorFlow

Many open source implementations for the model were found that could be used such as [8] [9] [10]. On the first attempt, we decided to use the TensorFlow [1] implementation available in the TensorFlow Github repository `https://github.com/tensorflow/models/tree/master/research/deep_speech`. TensorFlow is an open source library for numerical computation used for machine applications. It was developed and maintained by Google Brain. At the beginning it was intended for internal use only, however, on November 9, 2015, it was released under the Apache License 2.0.

3.3.1 Training on Google Colab GPUs

One of the earliest problems encountered was our inability to run the model locally, as it is a large model demanding very high computational resources. As a result, we turned to Google Colab, which is a free cloud platform used for machine learning education and research. It offers researchers the chance to run their applications using either a free GPU, or a cloud Tensor Processing Unit (TPU). A TPU is an Application-Specific Integrated Circuit (ASIC) optimized for performing high speed addition and multiplication operations, it was developed by Google for the purpose of speeding up machine learning applications.

Implementation Details

The implementation used TensorFlow Estimators, which is a high-level TensorFlow API that facilitates machine learning programming. Estimators encapsulate training, evaluation and prediction. They provide a training loop that controls building the graph, initializing variables, loading data, handling exceptions and creating checkpoints. For using Estimators, the data input pipeline should be separated from the model, thus making the model easier to work with different datasets. There are two types of Estimators: regular estimators, which work for both GPUs and CPUs, and TPU Estimators, which work for Google’s cloud TPUs. Since the implementation used normal estimators, we decided to run on Google Colab using a GPU which was a Tesla K80 with 12 GB memory.

[8] `https://github.com/noahchalifour/baidu-deepspeech2`
[9] `https://github.com/SeanNaren/deepspeech.pytorch`
[10] `https://github.com/tensorflow/models/tree/master/research/deep_speech`

Training Data, Process and Results

As an initial experiment to make sure everything works, we use only the Tuda-De data and stick to the train, dev and test sets which the authors provide. We use a model of 5 layers and 800 hidden size. We set the batch size to 2 in order to fit in the GPU memory. We use "Sorta Grad" and set the learning rate to 5e−5. We use Gated Recurrent Unit (GRU)s instead of LSTMs or vanilla RNNs. After few days, the training reached 3 epochs and a WER of 98% and a Character Error Rate (CER) was 43%. The issue was that training was notably slow that it was infeasible to proceed. The process was extremely time consuming and it would take months for the model to converge.

We tried to make the model's size smaller and used 3 layers with hidden size of 700. With this setting and using Tuda-De only, the model converged at 50% WER after more than a week. We made sure the model was working and able to learn, however, the training speed was considerably slow and it was infeasible to train on the whole datasets we collected.

3.3.2 Training on Google Cloud TPUs

In search of other alternatives, we tried to make use of TensorFlow Research Cloud Program. This program offers researchers free cloud TPUs in order to run their machine learning applications. We were offered 5 v2 cloud TPUs, each TPU had 8 cores with each core having 8 GiB of High Bandwidth Memory (HBM). The TPUs were available for use on Google Cloud Platform.

Migrating from Regular Estimator to TPU Estimator

The major obstacle was that the implementation used regular estimators; to overcome this it was necessary to migrate from normal estimators to TPU estimators. This process was a simple one that required little code changes. The main issue, however, was that models are compiled using Accelerated Linear Algebra (XLA) when using TPUs. XLA is a compiler that requires that tensor dimensions must be statically defined at compile time. The problem was that not all utterances had the same number of frames, and we could not set a specific length to pad up to it and truncate smaller than it. The challenge with truncating is that if we truncate after n frames, how do we determine the labels to discard after the n^{th} frame? The only solution was to pad the frames with zeros up to the maximum sequence length in the dataset, in this case, Tuda-De.

After padding to the maximum number of frames, we tried training using the same settings used on Google colab. Unfortunately, the TPUs ran out of memory and we could not proceed with the training. After searching for solutions to overcome this obstacle, one solution was to upgrade to TPU-v3 which has 16 GiB of HBM for each TPU core. The other was to use a TPU pod, which is a multiple TPU devices connected together with the workloads distributed across all devices. Unfortunately, both options were infeasible for us.

3.4 Deep Speech 2: PyTorch

In searching for other options, it was finally decided to abandon the TensorFlow implementation due to the problem of the TPUs requiring tensors of static dimensions. We turned, instead, to the PyTorch implementation of the model `https://github.com/SeanNaren/deepspeech.pytorch`. PyTorch is an open source machine learning library which was developed by Facebook's artificial intelligence research group. It is based on Torch library and was released under the Modified BSD license.

3.4.1 Training on Google Colab

In order to make sure the model was working and able to learn, we used only Tuda-De data. The model size was set to 3 layers and 700 nodes. All other settings and configurations were kept as in section 3.3.1. The training was started on Google Colab using a Tesla K80 GPU with 12 GB memory. The model was gradually improving, however, it was likewise considerably slow.

LSTM vs. GRU

All of our previous experiments were using GRUs as the hidden unit of the RNN layers. We decided to run two instances of training: one using 3 layers, 700 nodes, and GRUs. The other one using 3 layers, 700 nodes and LSTMs. All other settings were unchanged. After few epochs, the LSTM instance was performing much better.

3.4.2 Microsoft Azure Platform

At this point it was obvious there is a computational resources problem; Colab was very slow to use as it only used one GPU, and using the TPUs was not attainable. A cloud platform was needed in order to run the model and be able to iterate and experiment faster. For this purpose, we make use of Microsoft Azure's NC-series virtual machines. NC-series Virtual Machine (VM)s are armed with NVIDIA Tesla K80 GPUs and Intel Xeon E5-2690 v3 (Haswell) processors. We chose Standard NC12 size, with 12 CPUs and a total of 112 GiB memory. It has also 2 GPUs with GPU memory of 24 GiB and SSD storage of 680 GiB.

3.4.3 Training I: Using Tuda-De, M-AILABS, Common Voice

After getting access to adequate computational resources to experiment with, we were ready to run the training. It should be noted that we did not acquire all the data described in section 3.2.4 at once. It was a lengthy, time consuming process and some datasets were not included until late stages of the project. At first, we started with Tuda-De, M-AILABS and Common Voice. These datasets were used for the first 13 epochs, and then more data was added.

Speaker Independent Splitting

	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
Total	668.61h	34.63h	32.24h	5007

Table 3.2: Data used in the first 13 Epochs

With these 3 datasets at hand, we needed to split them among train, dev and test sets. One significant criteria for the splitting was that it had to be speaker independent. In other words, the utterances of one speaker can not exist in more than one of the three sets. The speaker independent splitting is crucial since we are interested in training a general speaker-independent ASR. Table 3.2 lists the datasets used in the 13 epochs of the training process along with the number of hours in each of the train, dev and test set. It is to be noted that we keep the dev and test set provided with Tuda-De and perform no changes on them. We also add all the utterances in M-AILABS to the train set, since 22 is the least number of hours per speaker. We do so in order not to sacrifice such large number of hours, and instead save it for training.

Epoch	WER	CER
1	61.286	16.690
2	46.4	11.998
3	40.663	10.190
4	34.519	8.456
5	32.162	7.801
6	29.811	7.368
7	28.487	6.903
8	27.907	6.927
9	26.624	6.543
10	27.005	6.662
11	26.004	6.413
12	25.658	6.320
13	25.499	6.282

Table 3.3: Training Summary for the First 13 epochs. The training data was

Training Process and Results

Using this data configuration, we start the training using a model of 5 layers, 800 nodes and LSTM units. A batch size of 20 is used and the learning rate is set to 3e−4. We follow the Sorta Grad technique and run the training for 13 epochs. The total training hours was 668.61 hours. At the end of epoch 13, the WER was 25.499% and the CER was 6.282%. The training summary is available in table 3.3 and figure 3.3

3.4.4 Training II: Adding all data

In order to be able to use all of the acquired datasets in the training process, we upgraded the VM to Standard NC24. This type has 24 CPUs and a total of 224 GiB memory, 4 GPUs with GPU memory of 48 GiB and SSD storage of 1440 GiB.

problem of adding SWC

We were faced with a problem when trying to use the German SWC dataset. The utterances were whole articles of very long durations that the memory would not fit them, even with the batch size set to 1. The dataset came with a segmentation file that included the alignments for the sentences. We used this segmentation file to cut the long `wav` files into small ones. Unfortunately, after manually analyzing a random sample of the generated files, a high percentage of it was found to be very faulty. Since we have no method of aligning the data ourselves like in [2], we refrained from using the SWC as we suspected it would deteriorate the overall performance.

Speaker Independent Splitting

In order to preserve the speaker independent splitting of the datasets after adding the movies data and CSS10, we add them all to the train set and refrain from adding any subsets to the dev or test. After adding all our data, we get a total of 727.3 training hours.

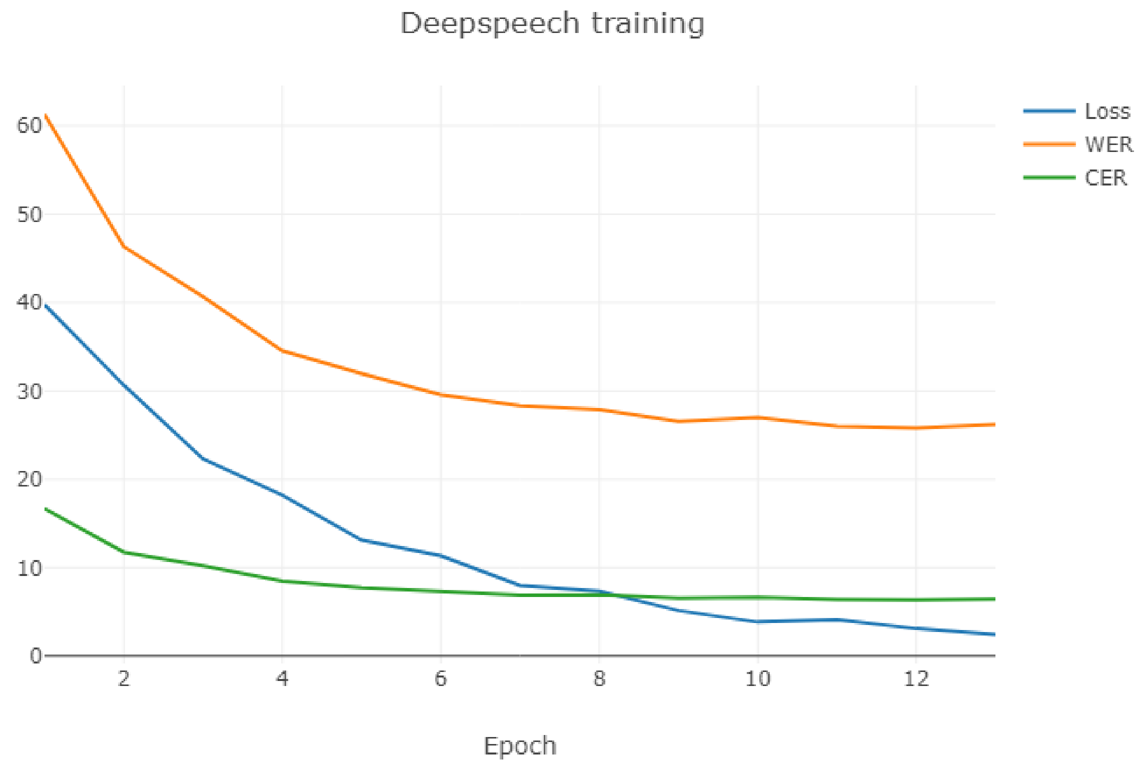


Figure 3.3: Training Summary after Epoch 13

	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: After adding Movies and CSS10 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 adopt 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 techniques were investigated. We discuss them in the following two 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 more than transcribed

Epoch	WER	CER
1	61.286	16.690
2	46.4	11.998
3	40.663	10.190
4	34.519	8.456
5	32.162	7.801
6	29.811	7.368
7	28.487	6.903
8	27.907	6.927
9	26.624	6.543
10	27.005	6.662
11	26.004	6.413
12	25.658	6.320
13	25.499	6.282
14	24.956	6.132
15	24.554	6.032
16	23.290	5.703
17	23.610	5.840
18	22.919	5.605
19	22.875	5.629

Table 3.5: Summary of the Training Process. Movies Data and CSS10 were added after epoch 13. Noise augmentation was also introduced after epoch 13

audio, we can train a language model on massive amount of text corpora to generate a powerful n-gram language model. For this task, we make use of KenLM, which is a Language Model Toolkit that is easy to use and can be used to generate n-grams language models of any order. The target was to collect abundant text corpora to experiment with language models. The datasets we found and used are listed here:

1. **German Wikipedia**
   We make use of the available German Wikipedia dump available for free download. The wikipedia is downloaded as one 20 GB xml file. We use a WikiExtractor library[11] to extract the Wikipedia into text files, which results in 56 folders, each containing 100 text files. Since KenLM requires that text data be in one file, with one sentence per line, we use spaCy[12] which is a Python library for NLP tasks. It supports many languages including German. We make use of the German model and run it on all of the Wikipedia text files. In order for the corpus to match the output of the ASR, we perform the same text cleaning performed on the ASR transcriptions: lowering all the letters, replacing numbers with their written form *i.e.* 139 is changed into einhundertfünfunddreissig, removing all punctuations and limiting all the vocab to letters from "a" to "z" plus the German umlauts ä, ö, ü. Any ß was replaced with "ss" and all the foreign characters were stripped out as well.

2. **5 Million Web Sentences**
   We also make use of the Leipzig Corpora Collection offered by Universität Leipzig. We make use of the data available in web section, and collect a total of 5 Million web sentences. The data was found to be one sentence per line and need not further processing other than the same text cleaning performed on the Wikipedia.

[11] https://github.com/attardi/wikiextractor
[12] https://spacy.io/

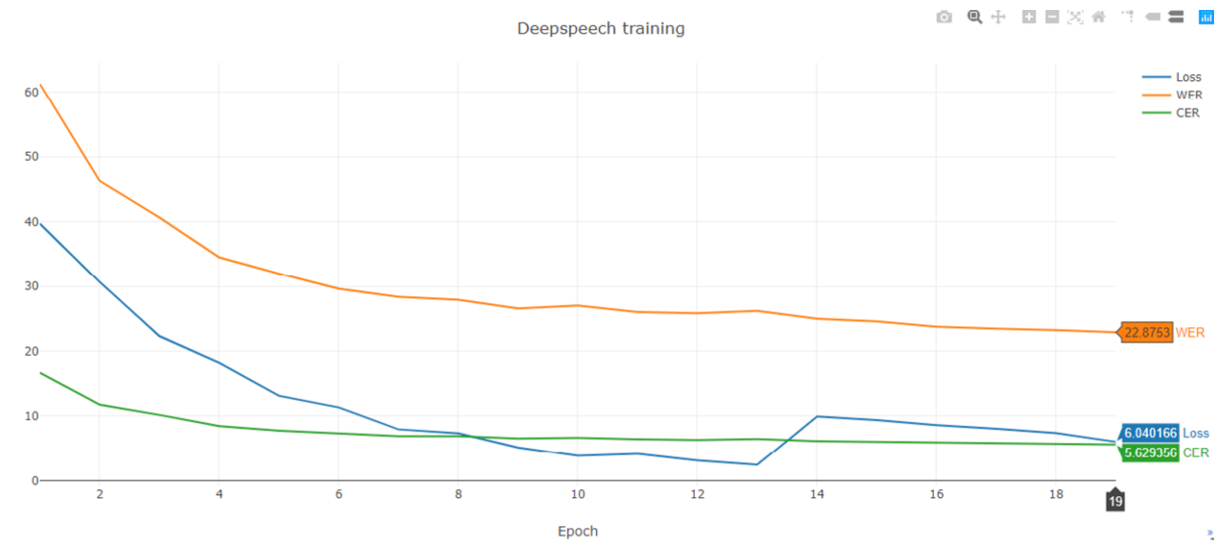


Figure 3.4: Training Summary after Epoch 19

3. **8 Million Maryfied Sentences**
   We make use of the data used for language modeling by ??, available for download here[13]. 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 for 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 ??

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

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

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

where the N-best scoring hypotheses outputs from the small language model are re-scored using a larger language model of higher n value *i.e.* 5 or 6 gram. The best scoring hypothesis according to the larger language model is then chosen to be the predicted transcription.

We expriment with various values for n for the smaller language model and the larger one. The different settings with their respective results are reported in details in the results section 4, however, our best WER was 21.633% and was achieved with a small language model of 3-gram built from all our text corpora, and a large language model of 5-gram built as well from all the text corpora. We used a beam-width of value 200, and we set alpha to 0.9, and beta to 0.2. Without re-scoring we managed to achieve 15.587% WER; this indicates that something is wrong with the setup or the implementation and further investigations need to be done.

3.4.7 Auto Correct With Transformer

In this experiment, we employ the method suggested in [39] by implementing an auto correct model that automatically corrects errors of the CTC-based network. The main idea is to train a model which takes as input at inference the predictions generated by the ASR model, and outputs the correct transcriptions.

Model Selection and Defining the Problem

We stick to the model proposed in the paper [39] which is the Transformer [34], however, we use the Transformer model implemented in tensor2tensor library. Tensor2Tensor (T2T) is a machine learning library developed and maintained by Google Brain that aims at making machine learning research accessible and easier for researchers.

We start off by installing the library and then turn to defining our problem. The library's general interface is to define a problem that can be solved using different models and/or hyper-parameters. There are pre-defined problems in the library, however, we define our own and we call it `asr_correction` problem. It is to be noted that our problem is very similar to the translation problems defined in the library.

Training Data and Results

For the training data, we use the movies data along with CSS10 data as these two datasets were only added after epoch 13 in the ASR training process, hence we can use the model's output at epoch 13 to transcribe these utterances. The training examples are made by pairing each transcribed output from the ASR with its ground truth. When transcribing these utterances using greedy decoding, we obtain a total of $75,000$ training examples. For the sake of increasing the number of training examples, we use a 5-gram model with beam-width 500, 0.9 alpha and 0.2 beta, and get the best 100 outputs. These outputs are paired with their corresponding ground truth. This technique increases our training examples by a factor of 100 resulting in around 7.5 Million training examples.

TODOOOOOOOOOOOOOOOOOOOOOOOOOOOO

We experiment with two hyper-parameter sets of the transformer: `transformer_base_single_gpu` and `transformer_big`. We train both instances for x steps, using the 750K examples. The transformer_big achieved x% WER, and x% CER, while the transformer_base_single_gpu achieved x% WER, and x% CER. We also train the transformer_big using the 75K training examples, however, more data got us an increase in performance by x%.

3.4.8 Comparing with the Hybrid Model

TODOOOOOOOOOOOOOOOOOOOOOOOOOO

For the purpose of comparing our system with the hybrid kaldi-based model [25] illustrated in section 3.2.1. We attempt to test the kaldi-based model on our test-set. We start by installing kaldi and Kaldi GStreamer server which is a real-time full-duplex speech recognition server. It is implemented in Python and based on GStreamer framework and Kaldi toolkit. Our test set is transcribed using the Kaldi GStreamer server and Kaldi's best model[14]. Then the output transcriptions are compared with the ground truth transcriptions in order to compute the WER and the CER.

TODOOOOOOOOOOOOOOOOOOOOOOOOOO After comparing the transcriptions, Kaldi's model achieved $x\%$ WER and $x\%$ CER on our test set. This means that our best model outperforms the kaldi-based model with $x\%$ improvement in WER and $x\%$ improvement in CER

TODOOOOOOOOOOOOOOOOOOOOOOOOOO

We also noticed that Kaldi-based model was tested on Tuda-De dataset excluding the realteck microphone since it's very low quality. We excluded all utterances recorded using this microphone, and tested our best model on the modified Tuda-De's test set. Our best model was after 19 epochs of training and coupled with a 5-gram language model decoding technique, beam-width value of x, using alpha and beta values of 0.9 and 0.2 respectively. We achieved $x\%$ WER and $x\%$ CER outperforming the kaldi-based model on this test set as well.

[14] http://ltdata1.informatik.uni-hamburg.de/kaldi_tuda_de/de_400k_nnet3chain_tdnn1f_2048_sp_bi.tar.bz2

3.5 Text Classifier Unit

In this section, we discuss the second part of our proposed system: the text classifier unit. The model for the text classifier was chosen to be Google's Bidirectional Encoder from Transformer (BERT) [11], illustrated in section 2.4.4 of the literature review. We illustrate the underlying theory behind our decision for using BERT over a large variety of both simple classifiers and general language-understanding models.

3.5.1 Why BERT?

A strong point of BERT over simple text classifiers is that we can make use of large amounts of unlabeled data to compensate for small labeled task-specific datasets. This is specially useful when we do not know details about the task-specific dataset, at least for the time being. (*i.e.* its size, its format, etc.) When using BERT as a classifier on Microsoft Research Paraphrase Corpus (MRPC)[15], which contains only $3,600$ training examples, the results were ranging between 84% and 88%. These results are very satisfying for such a small dataset. Since we do know the amount of data available for our task and given the results published for the MRPC dataset classification task, we assume that BERT would be of great help if the data available is of relatively small size. Moreover, we do not know the nature of the data available for our task. Using BERT, we can model our problem as a classification task, a question answering task, etc. Though we make the assumption that our problem is a text classification one, we utilize BERT for more flexibility. A further argument is the great importance for our task to get contextual information instead of performing shallow classification or using non-contextual learned embeddings or even using a model that shallowly concatenate left and right context. As discussed in section 2.4.4, BERT is a deep bidirectional model since it uses Masked LM (MLM) as one of its pre-training tasks. The next sentence prediction task also makes it model relationships between sentences, which is important for our task as well. In addition to this, fine-tuning a classifier on top of BERT is a fairly simple task. Google published two versions of pre-trained BERT, an English model, and a multilingual model that could be used for 104 languages, since our language of interest is German only, we suggest that training a German only model would be much more powerful for our intended purpose.

3.5.2 Implementation

Once we settled on using BERT for our text classifier unit, came the step of finding an implementation. For the sake of saving time and effort, we refrain from implementing our own model and search for a convenient implementation. Though there are many freely available implementations for Bert, we stick to the official Google implementation available in `https://github.com/google-research/bert` as it provides more flexibility. The implementation is based on TensorFlow library and makes use of the Estimator API. The open source code offers scripts for creating the training data, pre-training and finetuning, which minimizes the amount of work and modifications needed. As demonstrated earlier in the literature review, using BERT consists of two stages: pre-training and fine-tuning. We discuss the steps pursued during both stages.

3.5.3 Pre-training Data

For pre-training BERT, we needed large amount of unlabeled text data. The published pre-trained English models were trained on both the Book Corpus[16] (800M words), and the English Wikipedia[17]

[15]`https://www.microsoft.com/en-us/download/details.aspx?id=52398`
[16]`https://yknzhu.wixsite.com/mbweb`
[17]`https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2`

(2.5B words). The Book Corpus is not available for free download and we do now if it contains any German data, therefore we abstain from using it. Since we are interested in German, we make use of the German Wikipedia[18] alone (700M words - 52M sentences). It is convenient for use since the articles nature fits the next sentence task since most sentences are correlated. We considered using the sentences corpus provided by Universität Leipzig[19], however they were separate sentences and would not add much to the next sentence prediction task.

Pre-processing German Wikipedia

We start by downloading the German Wikipedia dump. The whole wikipedia is downloaded as one 20GB file in `xml` format. We use an open source wikiExtractor[20] in order to extract it into plain text files. The Wikipedia is extracted into $5,600$ text files split upon 56 folders, each 100 files in a folder. Since the input to our subsystem would be the output from the ASR unit, we perform some text cleaning in order to match the output of the ASR. The ASR outputs only lower-case letters ("a" to "z"), apostrophe ('), space, plus the German umlauts (ä ö ü). For this purpose, we replace all numbers with their written format *i.e.* 37 is turned into "siebenunddreissig". Any "ß" was replaced with "ss". All punctuation and foreign characters were stripped away, and only the allowed characters *i.e.* potential outputs of the ASR were kept in the corpus.

The implementation requires the input data to be in specific format: each file has to be a plain text file, with one sentence per line. Different documents must be delimited by empty new lines; this is essential for the next sentence prediction task. To get the data in the proper format, we use the spaCy[21] toolkit, which is a Python library for NLP tasks. It supports many languages including German, we make use of the German model and run it on all of the Wikipedia text files. There were many possible choices for the sentence tokenization task such as nltk[22], OpenNLP[23], etc. Nevertheless, we stick to spaCy since it is recommended by BERT's official github repository. The sentence tokenization is not flawless and by manually analyzing random samples of the files, it is found that many errors exist. We ignore the mistakes albeit and consider it as noise in the input data as it is advised in the repository to add a slight amount of noise to the input data. This is particularly beneficial to make the model robust to slightly different data during fine-tuning step. In addition, distinct articles are separated with empty lines.

Generating Vocab File

As previously illustrated, BERT uses word pieces instead of whole words. The process is done as follows: a vocab file containing sub-words is generated from the whole the pre-training data. This list of sub-words (referred to as vocab file), is used to tokenize the pre-training data and the fine-tuning data. Out of vocab sub-words are given a special `[UNK]` token. The official repository does not include code for learning new vocab, however, they refer to multiple alternatives such as Google's SentencePiece library[24] and tensor2tensor's WordPiece generation script[25]. Instead, we use an open source bert-vocab-builder[26], which is a modification to tensor2tensor's WordPiece generation script. The modifications introduced aims at making the generated vocab list more compatible with the

[18] https://dumps.wikimedia.org/dewiki/latest/dewiki-latest-pages-articles.xml.bz2
[19] http://wortschatz.uni-leipzig.de/en/download
[20] https://github.com/attardi/wikiextractor
[21] https://spacy.io/
[22] https://www.nltk.org/
[23] https://github.com/apache/opennlp
[24] https://github.com/google/sentencepiece
[25] https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/data_generators/text_encoder_build_subword.py
[26] https://github.com/kwonmha/bert-vocab-builder

tokenization script provided in BERT's repository. Changes include adding BERT's special tokens (`[SEP]`, `[CLS]`, `[MASK]`, `[UNK]`) to the vocab list, and denoting split words using "`##`" instead of "_".

A threshold of 800 is used in selecting the sub-words. *i.e.* any sub-word occurring in the training corpus less than 800 times will be excluded from the vocab list. Using this threshold, we end up with a vocab list of size $\approx 10,000$ sub-words. In later stages, all our data will be tokenized according to this vocab list.

Tokenization and Writing into TFRecord Format

After generating the vocab file, the pre-training corpus has to tokenized according to the produced vocab file. Google provided scripts for both tokenizing the corpus given a vocab file and creating the training data in the "TFRecord" format. TFRecord is a simple binary file format that many TensorFlow applications use for training data. In order to write the data into TfRecord format, the data has to be converted into sequence of byte-strings. Instead of concatenating all the Wikipedia's $5,600$ text files into one file, we modify the script to read the input files from a `.csv` file containing the paths for all the cleaned text files. This is done because all the examples of the input file will be stored in memory, so for the purpose of avoiding out of memory issues, we keep the files sharded and we modify the code to generate one TFRecord file for each 100 text files *i.e.* 56 total TFRecord files. A "`max_seq_length`" is set when creating the data, where sequences of lengths longer than this value are truncated and sequences of shorter lengths are padded. We choose this value to be 128. Another values set are the "`max_predictions_per_seq`" and "`masked_lm_prob`". The `masked_lm_prob` is the probability of masked tokens, we set this value to 0.15 *i.e.* 15%. The `max_predictions_per_seq` is the maximum number of MLM predictions per sequence. This value should be set to be `max_seq_length` * `masked_lm_prob`, therefore we set it to 20. It is to be noted that the script does not get this value automatically because this value has to be passed to both the data creation script and the pre-training script.

3.5.4 Pre-Training Process

Since the pre-training is a fairly expensive procedure, we required a cloud platform that offers services for high computational resources. We make use of the free cloud TPUs offered by TensorFlow Research Cloud Program. This program offers researchers free cloud TPUs in order to run their machine learning applications. We were offered 5 v2 cloud TPUs, each TPU has 8 cores with each core having 8 GiB of HBM. The TPUs were available for use on Google Cloud Platform.

We train with batch size of 896 sequences (largest batch size that could fit into TPU memory). (896 sequences * 128 tokens = $114,688$ tokens/batch). We set the `max_seq_length` and `max_predictions_per_seq` to 128 and 20 respectively since these values were used when creating the data. The `num_hidden_layers` and `num_attention_heads` are both set to 12, the `vocab_size` is set to $105,944$. Adam optimizer [24] was used with 1e$-$4 learning rate, β_1 and β_2 values set to 0.9 and 0.999 respectively. 0.01 L2 weight decay is used. Over the first $10,000$ training steps, learning rate warm-up is used and then linear decay of the learning rate. We A dropout probability of 0.1 is used on all layers. The activation function used is the "gelu" function [16], and the hidden size is set to 768. All of the model configuration (including vocab size) is specified in `bert_config` file. The total loss is the summation of the mean MLM likelihood and the mean next sentence prediction likelihood.

We train for $100,000$ steps, which is ≈ 16 epochs over our $700M$ words. (batch size 896 and sequence length 128) At the end of the training, the MLM task accuracy was 60% and next sentence accuracy was 98%

3.5.5 Fine-Tuning Data

After pre-training, came the fine-tuning stage. As discussed previously, our problem is modeled as a classification one. Therefore, we will be implementing a fine-tuning layer on top of BERT. For this purpose, we searched for task-specific datasets that could be used to fine-tune our pre-trained model. Specifically, we required a dataset with transcribed audio from the drivers and their corresponding dispatcher actions. However, we could not find any freely available datasets of this kind. We tried to obtain similar dataset from Trapeze, but unfortunately that was not possible. We, therefore, leave the task of dataset collection and fine-tuning the model on the collected dataset for future work.

10K German News Articles Dataset (10KGNAD)

Category	Train	Test	Combined
Web	1510	168	1678
Panorama	1509	168	1677
International	1360	151	1511
Wirtschaft	1270	141	1411
Sport	1081	120	1201
Inland	913	102	1014
Etat	601	67	668
Wissenschaft	516	57	573
Kultur	485	54	539

Table 3.6: 10KGNAD: Overview of train/test split

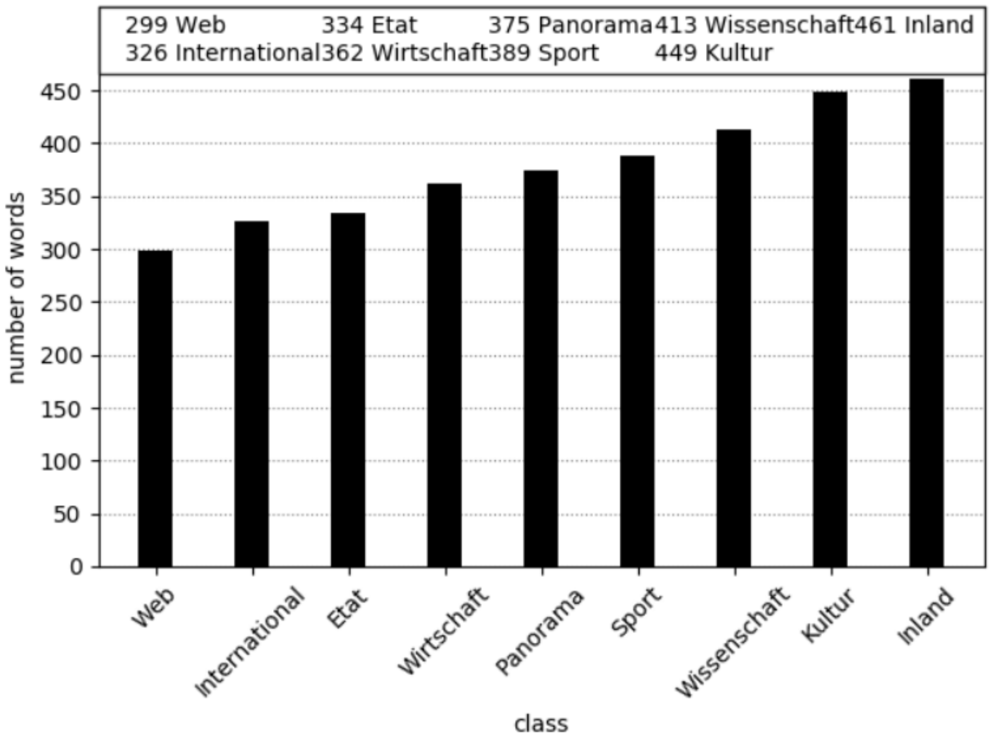


Figure 3.5: 10KGNAD: Articles per Class

We turn our attention, instead, to find a general classification dataset that could be used as a proof of concept that our BERT model unit could generate promising results as a classifier. One interesting freely available dataset is “10KGNAD”[27], which is a German topic classification dataset released under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. It constitutes 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 ?? and the train/test numbers given in table ??. 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

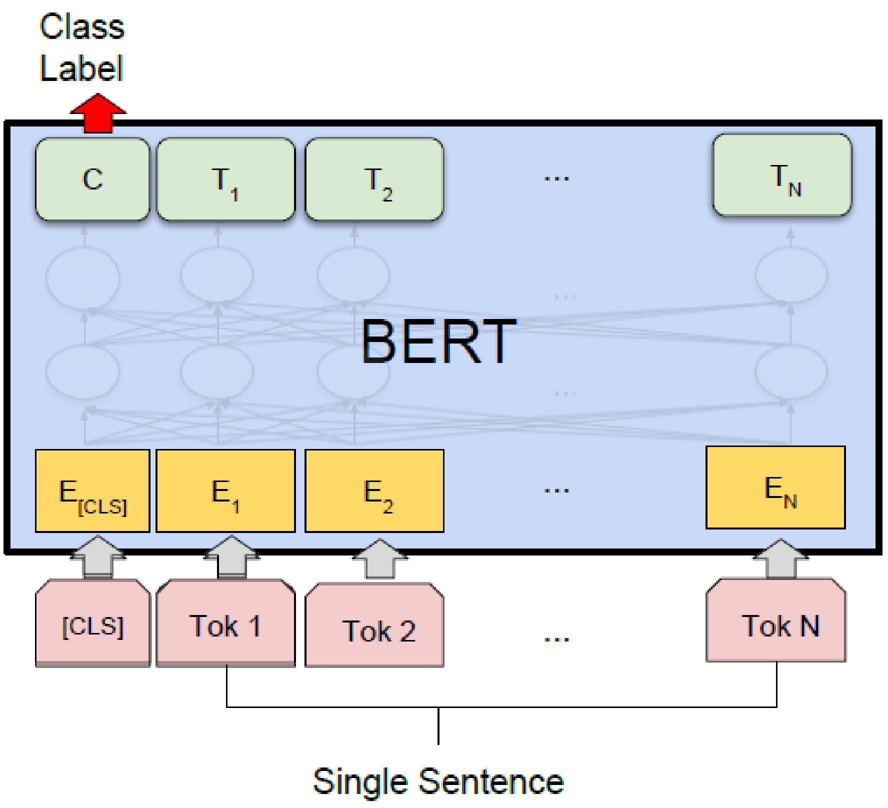


Figure 3.6: The task specific models are formed by building an output layer on top of BERT, so that small number of parameters need to be learned from scratch

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 ??. 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 og-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 file then fine-tune for 4 epochs using the same cloud TPUs used for pre-training. As for the result, we achieve an accuracy of 88% on the test set.

[27] `https://tblock.github.io/10kGNAD/`

Effect of Running Pre-Training on Fine-Tuning Data

It was advised in Google's github repository to run additional steps of pre-training on the task-specific dataset, starting from BERT's final checkpoint. This is specially beneficial if the dataset has different nature than the pre-training data. We experiment by running pre-training with the same settings reported in section **??**. Only the learning rate is changed, where we used 2e−5 instead of 1e−4 following the guidelines provided by the authors. After running $10,000$ steps of pre-training, we fine-tune again using the same settings used in section **??**. Running additional steps of pre-training on our task-specific dataset before fine-tuning gives us 2% improvement in the classification accuracy resulting in 90% on the test set.

Chapter 4

Results

In this chapter, the results for all the tests that have been held are reported in details.

4.1 Automatic Speech Recognition Unit: Deep Speech 2

4.1.1 ASR Evaluation Metrics

ASR Evaluation Metrics: WER CER

4.1.2 ASR Results

4.2 Text Classifier Unit: BERT

4.2.1 BERT Evaluation Metrics

BERT Evaluation Metrics: MLM Accuracy: likelihood how well the model scores in the predicting the masked words next sentence accuracy: how well the model scores in predicting the relationship between two sentence (is next, is not next) finetuning: classification accuracy:

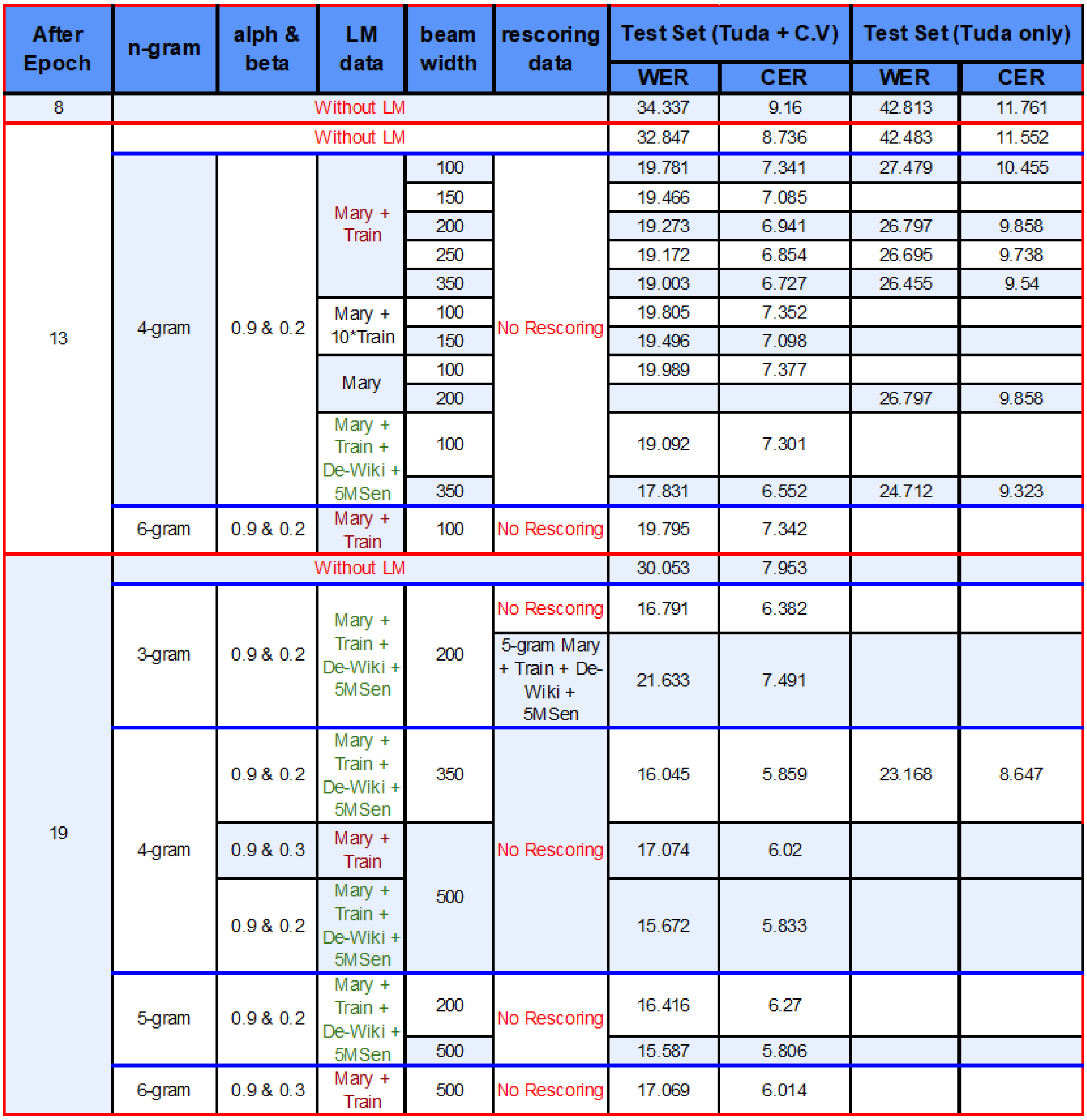
4.2.2 BERT Results

Masked LM Accuracy	0.60
Next Sentence Accuracy	0.98

Table 4.1: BERT Pre-Training Results

Classifier Result (fine-tuning only)	0.88
Classifier Result (extra steps of pre-training)	0.90

Table 4.2: BERT Fine-Tuning Results



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

Chapter 5

Conclusion, Limitations and Future Work

Conclusion

Appendix

Appendix A

Lists

NLP	Natural Language Processing
NLG	Natural Language Generation
NLU	Natural Language Understanding
ANN	Artificial Neural Network
FNN	Feed Forward Neural Network
MLP	Multi Layer Perceptrons
RNN	Recurrent Neural Network
BPTT	Backpropagation Through Time
BRNN	Bidirectional Recurrent Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
ASR	Automatic Speech Recognition
MFCC	Mel-Frequency Cepstral Coefficents
FFT	Fast Forier Transform
DCT	Discrete Cosine Transform
CTC	Connectionist Temporal Classification
HMM	Hidden Markov Model
CNN	Convolutional Neural Network
FSM	Finite State Machines
BERT	Bidirectional Encoder from Transformer
MLM	Masked LM
GMM	Gaussian Mixture Model
TDNN	Time-Delayed Neural Networks

SWC	Spoken Wikipedia Corpus
WER	Word Error Rate
CER	Character Error Rate
TPU	Tensor Processing Unit
ASIC	Application-Specific Integrated Circuit
HBM	High Bandwidth Memory
XLA	Accelerated Linear Algebra
VM	Virtual Machine
T2T	Tensor2Tensor
MRPC	Microsoft Research Paraphrase Corpus
10KGNAD	10K German News Articles Dataset

List of Figures

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[3] J Ba, J Kiros, and G Hinton. Layer normalization. arxiv. 2016.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] Leonard Baum. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a markov process. *Inequalities*, 3:1–8, 1972.

[6] Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[7] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[8] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[10] Povey Daniel, Ghoshal Arnab, Boulianne Gilles, Burget Lukas, and Glembek Ondrej. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number EPFL-CONF-192584, 2011.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[12] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[13] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1243–1252. JMLR. org, 2017.

[14] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM, 2006.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. 2016.

[17] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[18] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[20] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[21] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[22] MI Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. Technical report, California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science, 1986.

[23] Vlado Keselj. Speech and language processing daniel jurafsky and james h. martin, 2009.

[24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Benjamin Milde and Arne Köhn. Open source automatic speech recognition for german. In *Speech Communication; 13th ITG-Symposium*, pages 1–5. VDE, 2018.

[26] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. A time delay neural network architecture for efficient modeling of long temporal contexts. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[27] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[28] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[29] Stephan Radeck-Arneth, Benjamin Milde, Arvid Lange, Evandro Gouvêa, Stefan Radomski, Max Mühlhäuser, and Chris Biemann. Open source german distant speech recognition: Corpus and acoustic model. In *International Conference on Text, Speech, and Dialogue*, pages 480–488. Springer, 2015.

[30] Anthony Rousseau, Paul Deléglise, and Yannick Esteve. Enhancing the ted-lium corpus with selected data for language modeling and more ted talks. In *LREC*, pages 3935–3939, 2014.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

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

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[35] Alexander Waibel and Kai-Fu Lee. *Readings in speech recognition*, pages 393–404. Morgan Kaufmann, 1990. Phoneme recognition using time-delay neural networks.

[36] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[37] Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[38] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[39] Shiliang Zhang, Ming Lei, and Zhijie Yan. Automatic spelling correction with transformer for ctc-based end-to-end speech recognition. *arXiv preprint arXiv:1904.10045*, 2019.