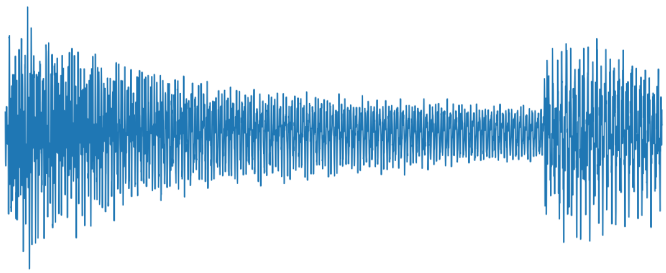
IT University of Copenhagen

Master of Science
Software Development: Advanced Computing

Thesis

Scalable Speech Recognition

Sebastian Benjamin Wrede
sbwr@itu.dk
Sebastian Baunsgaard
sebab@itu.dk



Supervised by

Pinar Tözün
Leon Derczynski

June 2, 2019

Abstract

This project describes an implementation of an *Automatic Speech Recognition* (ASR) system converting speech to text. It extracts *Mel* features, *Log Mel* features, and *Mel-Frequency Cepstral Coefficients* (MFCC) from sound and use them to train an *Acoustic Model* (AM) *Deep Neural Network* (DNN). The models are trained on two different hardware systems with four GPUs. The training process is benchmarked and optimized. Evaluation of the *throughput*, *latency*, and *accuracy* of the models is done and compared to other ASR systems. The best model implemented has a *Word Error Rate* (WER) of 10.5 and a latency shorter than the duration of the input making it appropriate for real-time applications.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Speech is one of the most prevalent ways of communicating among humans. It is a fast and convenient way to convey information from one person to another and technology to transmit it such as telephones and radio has been developed long ago. The technology has greatly influenced how people communicate providing immense value all over the world. The widespread use has also posed challenges in how to interpret, store and search speech data in computer systems. The challenges have given rise to the research area of *Automatic Speech Recognition* (ASR) that tries to improve the human-computer and human-human interactions by developing computer systems that automatically recognize speech and infer meaning from it [YD15, p. 1].

ASR has been a research area a long time and the recent years' computational power improvements have strengthened the academic interest in the area. This have lead to the great increase of mobile devices with virtual assistant systems using ASR. Mobile devices and virtual assistants such as Amazon's Alexa [Ama], Apple's Siri [App], Microsoft's Cortana [Mic] and Google Assistant [Goo] use ASR for voice search, message dictation and a range of other daily tasks. This type of ASR can be handled locally on the device with compressed low-latency models built for instance with TensorFlow Lite [Ten19a] [ZK18, pp. 34-35].

Another type of ASR system take a centralized approach to ASR, where speech is converted to text on a server. The text can then be retrieved and analyzed later. This type of system is relevant in most organizations dealing with large amounts of speech such as hospitals, where medical records are dictated by doctors [Joh+14], and call-centers, where customer data needs to be easily accessible to improve customer service [Mis+05].

A benefit of such centralized systems is the larger capacity to compute the results utilizing hardware acceleration. Most calculations in neural network models are small and independent, hence a significant performance improvement can be achieved by parallelizing the systems on specialized multithreaded hardware such as GPUs [SBS05; RMN09].

When a machine learning model is deployed in a centralized server, new challenges of processing incoming requests arise. The system should be able to handle a large number of concurrent requests with an acceptable latency and be able to handle more concurrent requests when adding new or more hardware resources. The ability to handle a large number of concurrent requests with an acceptable latency is referred to as *scalability* in this project.

In the cross field of hardware acceleration and ASR, this master's thesis has the research question:

How do we implement and efficiently train a centralized deep learning system converting speech to text in a scalable manner?

A source of inspiration for the system is the two ASR models developed by the Baidu Research group, *Deep Speech 1* [Han+14] and *Deep Speech 2* [Amo+15], and later the unpublished model called *Streaming Multi-Layer Truncated Attention Model* (SMLTA) [Gro19] also by the Baidu Research group. Other ASR systems are also investigated during the project, such as *Cold Fusion* [Sri+17], *Listen, Attend and Spell* (LAS) [Cha+15], *Unidirectional LAS* [Chi+17], *Hard Monotonic Attention* [Raf+17] and many more.

Using the TensorFlow framework in Python, a number of different deep learning models inspired by Deep Speech 2 [Amo+15] are implemented and trained using two different GPU clusters. The performance of the cluster is important when training the models since it requires an extensive amount of computation to train. For this reason, performance benchmarks are conducted and the system adjusted accordingly to reduce the amount of training time.

When a model has been trained, it is served using TensorFlow Serving [Ten19b] and evaluated based on latency, throughput and output quality of the served

model.

This thesis starts with a background chapter 2 describing the basics of ASR technology and the TensorFlow framework. Chapter 3 describe related work in the field of ASR and mention background papers on GPU neural networks training as well as GPU systems available. Chapter 4 describes the implementation of the system in TensorFlow and how the models are served. It also cover the data used for training and performance evaluation as well as the system's hardware configurations. Chapter 5 shows the performance of different models during training on the GPUs measuring how the different configurations are affected by different hyper parameters. Chapter 6 shows the results of the trained models based on the latency, throughput and quality of the inferences. Chapter 7 mentions further work that could be done to improve the system. Finally, chapter 8 provides a summary and conclusion of the entire thesis.

2 Background

This chapter describes the theory involved in building a modern neural network based automatic speech recognition system. First, the fundamentals of digital sound is described to enable extraction of features from sound. Then the deep learning theory involved in the system is explained from the basics of Feed Forward Neural Networks (FFNN) to Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). Optimization algorithms for training the model are also explained including Stochastic Gradient Descent (SDG), AdaGrad and AdaDelta. Additionally, Activation Functions, Back-Propagation, Batching and Regularisation are explained. The end of the chapter has a brief description of TensorFlow, which is the framework used for this project. All theory is explained to construct models for converting sound to text in this project. Section 1, section 2 and section 3 has been adapted from our Thesis Preparation report [WB18].

1 Analog & Digital Sound

Speech is a specific type of sound. Sound is created from everything that moves by pushing and pulling air, or any other matter, causing pressure variations. Those pressure variations can be called sound waves. Waves can be represented as a continuous oscillating function which is also called an *analog function* or an *analog signal*. Sound represented like this can be called *analog sound*.

Analog sound can be converted to digital sound through an Analog to Digital Converter (ADC) [Bur+11].

Digital sound is the quantization of an analog sound that takes discrete measurements of the sound wave. The frequency at which the samples are taken is called the *sampling frequency* or *sampling rate* which is measured in Hertz (Hz) [KS16, p 38].

The Nyquist-Shannon sampling theorem states that to capture a frequency the sampling rate must be double that frequency. This means that if the sampling frequency is f_{sample} then the maximum sound frequency that is captured is $\frac{1}{2} * f_{sample}$ [Bur+11, Section 2.3] [KS16, p 38].

The sampling frequency of CD's are usually 44,100 Hz which results in a maximum captured sound frequency of 22,050 Hz. The human ear is only able to capture frequencies of up to around 20,000 Hz while young, hence the sampling frequency of CD's makes it possible to capture all sound that humans are able to hear [Bur+11, Section 2.3]. The sampling frequency of phone conversations over the DS0 and E0 network is 8,000 Hz resulting in a maximum captured sound frequency of 4,000 Hz which is sufficient for oral communication among humans, but is also the reason why people sound different on the phone [ITU88].

When constructing this digital sound, a *low-pass filter* that only allows frequencies below a given threshold is used at half the sampling rate (adhering to the Nyquist-Shannon sampling theorem) to avoid aliasing. Aliasing is when high frequencies are sampled as lower frequencies which impairs the quality of the sampled sound [Gal15] [Bur+11, Section 2.3].

Microphones capture all sound, and are therefore susceptible to noise from the background. This can be reduced using noise cancellation algorithms such as for instance Power Level Difference (PLD) that given two microphones on a mobile phone, in which one is used for speech, cancels out the other microphone's observed signals [Zha+]. More advanced algorithms are able to cancel out more noise and the algorithms keep improving. They are applied in most modern phones [Tri17].

2 Speech Properties

Multiple properties of speech define the difficulty of transferring the speech to text. Here are the three main properties as defined in "A Review on Speech Recognition Challenges and Approaches" by Vimala and Dr. Radha [VD12]:

Types of Speech Utterance How are the words uttered?

Isolated words, takes only a single utterance at a time which requires a pause between each utterance; *connected words*, allow some utterances to be connected which means there are only short pauses between the utterances; *continuous speech*, where the speaker can speak almost naturally; and *spontaneous speech*, where the system should be able to handle all natural speech including false-starts and mispronunciations.

Speaker Model Who's speech is the system able to understand?

The speaker model is important for understanding the spoken word. The extremes are *speaker independent*, where the algorithm is independent of who and where the speaker is. The other extreme is *speaker dependent*, where it requires a specific speaker and a specific setting.

Vocabulary How many different words and phrases do the system know?

The size of vocabulary is important when converting speech to text. Small vocabularies such as digits to text are easier than larger vocabulary sizes. *Out-of-vocabulary*, is when the system needs to map unknown words, and is a difficult challenge.

The ideal system would be able to in real-time recognize with 100% accuracy all words spoken by any person, independent of vocabulary size, noise, speaker characteristics or accent [YD15, p. 5]. All speech datasets can be characterized on a scale as in fig. 2.1.

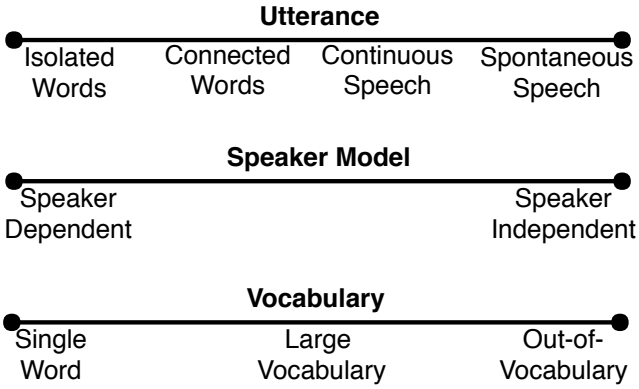


Figure 2.1: Speech Properties.

3 Fundamentals of ASR

This section describes the fundamental components of ASR systems. The basic components are *Feature Extraction* (FE), *Acoustic Model* (AM) and *Language Model* (LM) [YD15, p. 4]. Each of these components have a separate purpose in the ASR system, however an ASR system is not required to include all the components separately. Recent research has moved towards end-to-end ASR models based on deep learning [Han+14; Amo+15; Cha+15; Bat+17; GSW19], which is described in chapter 3.

This related work use some variation of FE based on the same techniques described in this report, but they do not cover it in depth. The main argument against the FE used in this report and many other similar systems is that they fundamentally only rely on an analysis of the frequencies in the sound [Lyo17, pp. 13 + 16-18].

3.1 Feature Extraction

To enable ASR we need to extract features from sound. This is referred to as *Feature Extraction* (FE). A lot of different features can be extracted from both the time domain and the frequency domain, such as Amplitude, Root-Mean-Square Energy, Zero-Crossing rate, Band Energy Ratio, Spectral Centroid, Bandwidth and Spectral Flux [KS16, p 40 - 50]. In our case we only use the Amplitude but there are potential benefits from merging features with some of the other types.

Time domain is a representation of sound with the amplitude over time which in graph representation is time on one axis and amplitude on the other axis. This is shown in the top graph of fig. 2.2. *Frequency domain* is a representation of sound with intensity over different frequencies for a given time interval which in graph representation has frequencies on one axis and intensity on the other axis [Bur+11, Section 3.1] [Lyo17, p. 16]. This is shown in the second graph of fig. 2.2.

Fourier Transformation

To move sound from the time domain into the frequency domain, *Fourier Transformations* (FT) and framing is applied. Any frame of sound can be approximated using a *Fourier Series* as defined in the following equation:

$$f(x) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(n\ x) + b_n \sin(n\ x)) \tag{2.1}$$

It states that the sound wave in a frame can be described with an infinite sum of cosine and sine functions over input time x. The sum of functions can be changed by adjusting the coefficients a_0, a_n and b_n which are called the *Fourier coefficients* [Ste13]. FT converts the signal from time domain into the frequency domain by using the Fourier coefficients to produce complex values for each frequency band representing the magnitude of that frequency [JM09, p. 300].

When the input signal is discrete, as it is when working with digital audio, a *Discrete Fourier Transform* (DFT) can be used to reduce the infinite limit of the FT [KS16, p. 40]:

$$X_m = \sum_{k=0}^{K-1} x_k \cdot e^{-\frac{2 \cdot \pi \cdot i}{K} \cdot k \cdot m} \tag{2.2}$$

K is the number of samples in the frame, m is an integer ranging from 0 to $k-1$, x_k is the amplitude of the k'th sample of the given frame and X_m is a complex number from which the frequency domain is obtained by computing it over different time domain frequencies [KS16, pp.40-41].

An algorithm to perform the DFT was developed by Cooley and Tukey with a running time of $O(K \cdot \log K)$. This is commonly referred to as *Fast Fourier Transform* (FFT) although several variants of FFT exist. The FFT requires the number of samples in each frame K to be a power of two.

When working with waves that have varying frequencies over time, a *Short-Time Fourier Transform* (STFT) [KS16, p. 41] is used. An STFT computes DFTs over different frames of the input signal. The STFT has a *frame size*, specifying the number of samples in each frame, and a *frame step*, specifying the number of samples between each frame. The frame step enables each frame to overlap, to enable finer granularity in the features extracted or make space between frames to reduce the amount of calculations.

The DFT result of each frame constructed by the STFT can be concatenated into a *magnitude spectrogram* [Lyo17, p 15] that contains all FTs for all the frames on a timeline. An example of a magnitude spectrogram can be seen in the second figure from the top in fig. 2.2 with the title "Frequency".

Mel-Frequency Cepstral Coefficients (MFCC)

Human hearing is not the same as the machine's hearing which is why there are multiple transformation functions that transform the frequency domain into something that based on empirical studies simulates how humans perceive sounds. The two most well known ones are the *Bark* and the *Mel* scales [KS16, p 53 - 56]

These scales enhance the frequencies that humans perceive well. In our studies we are going to focus on and use the Mel scale as seen in eq. (2.3) where m is the Mel-frequency of Hertz-frequency f [KS16, p. 54].

$$m = 1127 \cdot \log\left(1 + \frac{f}{700}\right) \tag{2.3}$$

Furthermore if the logarithm is applied yet another time to the Mel output it becomes *Log Mel*, and can be seen in the fourth graph of fig. 2.2. The Log Mel result is used to calculate the *Mel-Frequency Cepstral Coefficients* (MFCC). It is a combination of converting the Hertz frequencies to Mel frequencies, then to Log Mel and finally applying another *Discrete Cosine Transform* (DCT). The MFCC returns a number of *Cepstral Coefficients* that represent the sound in the given frame [KS16, p. 55].

Only the first 13 dimensions of the the MFCC is used. The MFCC plot in fig. 2.2 is reduced to the 13 dimensions and have removed the first dimension. The first dimension is removed from the plot because its values have a much larger range than the other dimensions making details in the plot almost invisible. All 13 features are typically used as input.

3.2 Acoustic Model

An *acoustic model* (AM) takes as input the features of the sound. These sound features could be the previously mentioned Amplitude, Frequency, Mel, Log Mel or MFCC, and outputs AM scores based on the relations between acoustics and phonetics [YD15, p. 4]. This AM score can subsequently be used in combination with the Language Model (see subsection 3.3) to determine spoken words. In this section, two different approaches to constructing an AM are considered. The first approach, called *GMM-HMM*, has been state-of-the-art through the 90's and 00's, but it is now outperformed by Neural Network-based approaches such as *CD-DNN-HMM* [YD15, p. 5] [Sai+15] and the systems described in chapter 3.

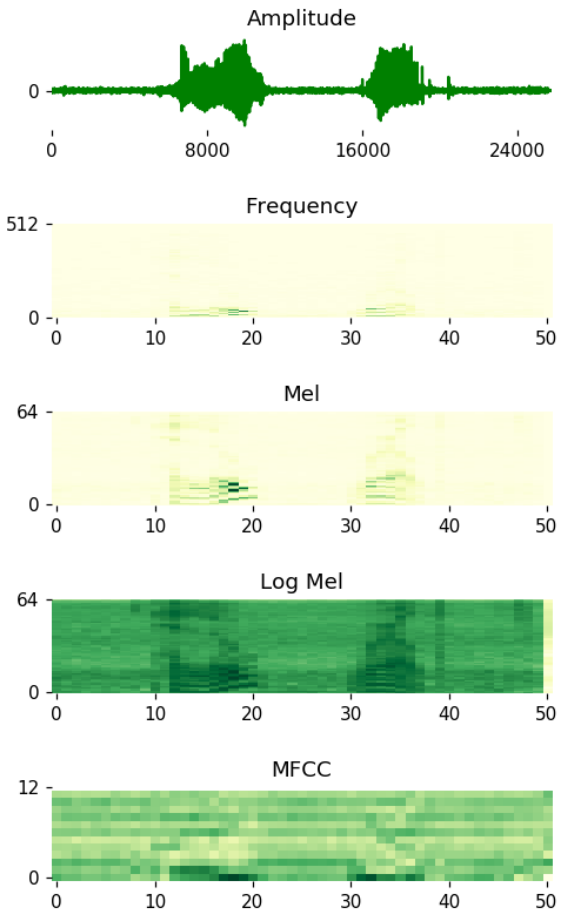


Figure 2.2: Feature extraction of sound saying *"hello world"*.

GMM-HMM

GMM-HMM is an abbreviation of *Gaussian Mixture Model - Hidden Markov Model*. It uses a GMM to represent distributions and a HMM to capture sequence information [YD15, p. 19].

One of the challenges of using GMM as an acoustic model has been the evaluation speed, the amount of training data required to avoid overfitting and the complex representation of otherwise simple systems [YD15, pp. 19-20].

A Hidden Markov Model (HMM) can be used in relation to the GMM to capture sequence information. A HMM consists of states, distributions of the states and probabilities of transitions between the states [Rab89; RJ86].

The relation between the output sequence and speech is that the states in a GMM-HMM are typically associated with a part of a phone in speech. However, GMM-HMM has a weakness of modeling speech dynamics which (in spite of several extended HMMs) is part of the reason that its success has been surpassed by Neural Network-based methods [YD15, pp. 43-44].

Neural Network Methods

Several methods using Neural Networks have been invented, one of the earliest being the DNN-HMM hybrid system (DNN is an abbreviation of Deep Neural Network).

DNN-HMM uses the DNN to calculate the probabilities of the input features representing different phones, hence each output neuron of the DNN would represent a possible phone and the output value from the neuron would represent the probability that the given frame represents that phone. The output of the DNN is given to the HMM to represent the dynamics of the speech signal which is one of the disadvantages of this model as it then has the same weaknesses of modeling speech dynamics as described in the previous subsection [YD15, pp. 99-100].

Other approaches combining DNN and HMM also exist, such as Context Dependent-DNN-HMM (CD-DNN-HMM) which takes several feature frames as input and uses senones as output from the neural network instead of monophones. A *senone* is a component of a *triphone* which is a context-dependent description of a monophone [CMU]. This means that using senones as output of the DNN makes the representation of sound more detailed. The use of the CD-DNN-HMM

has three improvements over other models mentioned earlier in this paper: (1) using deep neural networks, (2) using long window frames instead of separate frames, and (3) using senones as output instead of monophones [YD15, p. 106]. The Word Error Rate (WER) is improved in CD-DNN-HMM compared to GMM-HMM systems according to several studies [YD15, p. 303].

3.3 Language Model

The acoustic model outputs an AM score which can be combined with the Language Model (LM) to generate an output word sequence [YD15, p. 4]. The LM calculates probabilities of which words are uttered next based on prior words in a word sequence.

If we look at a sequence of n words as $S = w_1, ..., w_n$, then there is an associated probability of that specific sequence of words being: $P(S) = P(w_1, w_2, ..., w_n)$. Applying the chain rule this becomes the probability of the first word times the probability of the second word given the first and so on, as in: $P(S) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)...P(w_n|w_1, ..., w_{n-1})$.

The foundation of many language models is the Markov assumption. This assumption says that an approximation of a probability can be given by observing a limited number of previous observations K:

$$P(w_n|w_1, ..., w_{n-1}) \approx P(w_n|w_{n-K+1}, ..., w_{n-1}) \quad (2.4)$$

The LM can be constructed in several different ways. In this section, we will go through N-Gram models and an NN-based model.

N-Gram Language Models

In the N-Gram model, probabilities of sequences of words are calculated based on a large corpora of text. An *n-gram* is a a sequence of n words where n is an integer. A 2-gram is called a *bi-gram* and a 3-gram is called a *tri-gram*. If a word w_i is predicted using n-grams, the probability is calculated based on the $K = n - 1$ preceding words as in Equation 2.4 [Spe15].

Neural Network Language Models

Another approach is using neural networks, usually in the form of an RNN or an LSTM. By converting each word to a vector of numbers called a *word embedding*, the NNs can take the word embeddings as input and be trained to output a probability distribution of the next word in the given sequence of words. The neural network approach is able to model a longer range of dependencies compared to the classical statistical approaches such as the N-Gram model [Koe16].

4 Deep Learning

An acoustic model can be built using concepts from deep learning. The basic concepts are described in this section, beginning with the different types of models in section 4.1 followed by descriptions of different activation functions used in the layers in section 4.2. Back-propagation, loss functions, and batching are then described in section 4.3, section 4.4, and section 4.5 respectively. Different algorithms for optimizing the model are presented in section 4.6 and regularization strategies for improving the generalization of a model is defined in section 4.7.

4.1 Model

Deep Feed Forward Network

The basic type of deep learning model is a *Deep Feed Forward Network*, which is also called *Feed Forward Neural Network* (FFNN) and *Multilayer Perceptrons* (MLPs) [GBC16, p. 163]. We will refer to it as FFNN in this report. An FFNN is composed of a number of different functions which each represent a node in the network. It can be seen as a directed acyclic graph where the nodes are the functions and the output of each function is a directed edge. The nodes are organized in layers and by combining a large number of layers, the depth of the network is increased. This depth of the network is the origin of the term *Deep Learning* [GBC16, p. 164].

The first layer of an FFNN is called an *input layer*, the final layer of the FFNN is called the *output layer* and the other layers are called *hidden layers*. The hidden layers consists of *hidden units* which accept a vector of inputs x and compute an output h with the following:

$$\boldsymbol{h} = g\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right) \quad (2.5)$$

W is a weight matrix also referred to as the kernel and b is the bias. W and b are both initialized and changed during training of the model. The initialization of W and b could be random, although this tend to perform poorly in deep neural networks [GB10, p. 1]. Instead, a *Xavier Initialization* could be used. The Xavier Initialization scales all gradients in all layers according

to a uniform distribution within a range [GB10; Ten19q]. Training of models will be described in more detail in section 4.6. The function g in eq. (2.5) is an element-wise nonlinear function called an *activation function* which will be described in section 4.2 [GBC16, p. 187].

The output of one layer becomes the input of the next layer. Layer number l of an FFNN can be formally defined as [GBC16, p. 191]:

$$\boldsymbol{h}^{(l)} = g^{(l)}\left(\boldsymbol{W}^{(l)}\boldsymbol{x} + \boldsymbol{b}^{(l)}\right) \tag{2.6}$$

The next layer $\boldsymbol{h}^{(l+1)}$ of the FFNN is then defined as:

$$\boldsymbol{h}^{(l+1)} = g^{(l+1)}\left(\boldsymbol{W}^{(l+1)}\boldsymbol{h}^{(l)} + \boldsymbol{b}^{(l+1)}\right) \tag{2.7}$$

It is most common for layers to be fully connected which means that all hidden units in layer $\boldsymbol{h}^{(l)}$ are connected to all units in the next layer $\boldsymbol{h}^{(l+1)}$. However, it is also possible to have fewer connections so that only a subset of units in $\boldsymbol{h}^{(l)}$ are connected to a subset of units in $\boldsymbol{h}^{(l+1)}$. The decreased number of connections decreases the parameter size and consequently the computation required [GBC16, p. 196]. Layers are most commonly connected in a chain so that layer $\boldsymbol{h}^{(l)}$ are connected to $\boldsymbol{h}^{(l+1)}$ which are connected to $\boldsymbol{h}^{(l+2)}$. It is also possible to skip layers so that $\boldsymbol{h}^{(l)}$ is connected directly to $\boldsymbol{h}^{(l+2)}$. This makes it easier for the gradients to flow from the output layer to the input layer [GBC16, p. 196]. Gradients will be described in section 4.3.

Convolutional Neural Network

A *Convolutional Neural Network* (CNN) is a special kind of neural network based on a mathematical operation called a *convolution*. The convolution operation is used widely in signal processing [Wil18; Smi02], but the convolution operation as used in signal processing and pure mathematics does not correspond exactly to the use in neural networks [GBC16, p. 321].

The convolution operation is of the form:

$$s(t) = (x * w)(t) \tag{2.8}$$

where $s(t)$ is called the *feature map*, x is a function referred to as the input and w is the kernel [GBC16, p. 322]. A convolution "blends" the function x with the function w with an integral that expresses the amount of overlap of the two functions. The kernel w is in this way used to smooth the function x over the time t by using the function w [Wil18]:

$$s(t) = \int_{-\infty}^{\infty} x(\tau)w(t-\tau)d\tau \tag{2.9}$$

By doing so, it is possible to create an output that for a time t in the signal is determined by the part of the signal immediately surrounding t by designing w to weigh the parts of the signal closer to t higher. This could for instance be done if the function w is a Gaussian function then the output will be based on the Gaussian distribution around t [Wil18].

When working with discrete signals, $s(t)$ becomes a sum of multiplications:

$$s(t) = (x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t-\tau) \tag{2.10}$$

where x and w are defined on discrete values. This means that if the input x and the kernel w are two-dimensional and discrete, then the convolution can be defined as [GBC16, p. 323]:

$$S(i,j) = (W * X)(i,j) = \sum_m \sum_n X(i-m, j-n)W(m,n) \tag{2.11}$$

This can be perceived as a matrix multiplication if X and W are matrices and this is exactly how it is used in CNNs. The kernel is slided over the input and for each step a matrix multiplication is performed. For instance in fig. 2.3, convolutions are performed with a 3×4 input and a 2×2 kernel producing a 2×3 output.

The spatial size of the output of a CNN is given by [Kar19]:

$$h_{size}^{(o)} = \frac{W - F + 2P}{S} + 1 \tag{2.12}$$

where W is the input size, F is the *receptive field* size, S is the *stride* and P is the *zero-padding*. The units $x_r^{(l)}$ in one layer that is connected to a specific unit $s_r^{(l+1)}$ in the next layer are referred to as the receptive field of $s_r^{(l+1)}$. An example of this is given in fig. 2.4 where x_2, x_3 and x_4 is the receptive field of s_3.

The stride defines how many steps the kernel is moved with each slide. Given a stride z, a convolution with stride can be defined by extending eq. (2.11) [GBC16, p. 338]:

$$\begin{aligned} S(i,j) &= (W * X)(i,j) \\ &= \sum_m \sum_n X(i - m \times z, j - n \times z)W(m,n) \end{aligned} \tag{2.13}$$

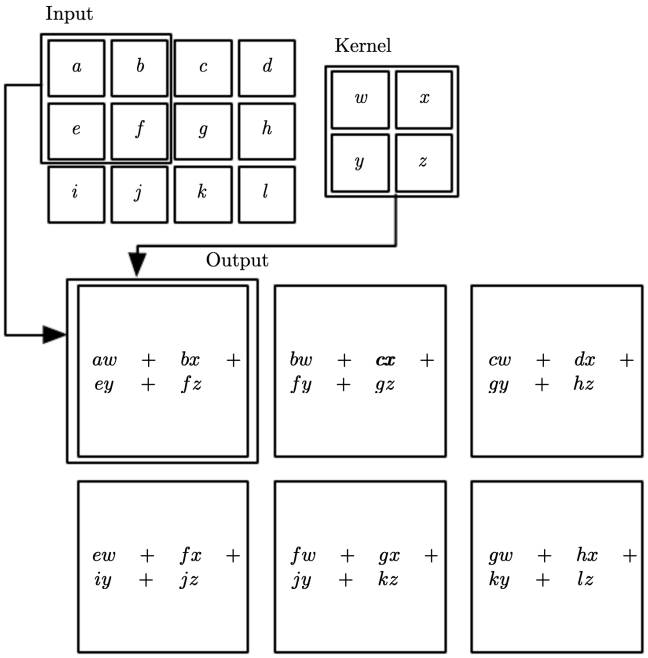


Figure 2.3: An example of a 2-D convolution from Goodfellow, Bengio, and Courville [GBC16].

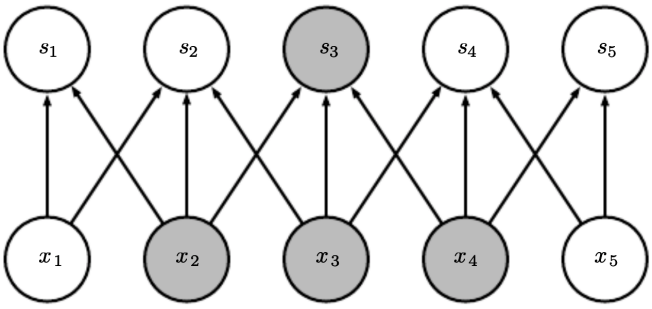


Figure 2.4: Example of receptive field and connectivity from Goodfellow, Bengio, and Courville [GBC16].

A larger stride will decrease the output size according to eq. (2.12).

The input can be padded with any number of zeros, which is referred to as zero-padding. A zero-padding enables the kernel to move to the edges of the input matrix. Further it is used to control the spatial output size as specified in eq. (2.12) [Kar19]. If zero-padding is not used, the size of the next layer will be smaller than the current layer [GBC16, p. 338]. This can also be seen in fig. 2.3 where the size is decreased from 4×3 to 3×2.

CNNs have several advantages including sparse interactions and parameter sharing. Sparse interactions occur when the kernel is smaller than the input. With fully connected layers of size m and n, algorithms performing the matrix multiplications require $m \times n$ parameters whereas with only k connections this is reduced to $k \times n$ [GBC16, p. 326]. The concept of parameter sharing means that every weight in the kernel is reused several times, one time for each element of the input except the edge elements, which is in contract to the FFNN where the weights in the weight matrix are only used once. A lower number of weights reduces the running time of the back-propagation because the back-propagation updates all weights [GBC16, p. 328]. A reduced number of weights leads to a reduced running time.

Recurrent Neural Network

A *Recurrent Neural Network* (RNN) is, in contrast to an FFNN, a directed cyclic computational graph. An RNN can take arbitrary length input sequences and apply the same function to all elements of the sequence thereby avoiding the need to train a separate model for all input elements. This is another type of parameter sharing which improves the efficiency of training the model.

An RNN takes a list of input vectors $x_{1:n} = x_1, ..., x_n$ and generates a list of output vectors $y_{1:n} = y_1, ..., y_n$ and state vectors $s_{1:n} = s_1, ..., s_n$ by recurrently applying a function R on one of the input vectors x_i and a previous state s_{i-1} to obtain a state s_i on which a function O is applied to produce an output vector y_i. This is described in eq. (2.14), where s_0 is an initial state that can be defined using the Xavier Initialization as described in section 4.1.

$$\begin{aligned} RNN\left(\mathbf{s_0}, \mathbf{x_{1:n}}\right) &= \mathbf{s_{1:n}}, \mathbf{y_{1:n}} \\ \mathbf{s_i} &= R\left(\mathbf{s_{i-1}}, \mathbf{x_i}\right) \\ \mathbf{y_i} &= O\left(\mathbf{s_i}\right) \end{aligned} \tag{2.14}$$

A visualization of a general RNN from Goldberg [Gol16, pp. 46-47] [GH17, pp. 164-167] is seen in fig. 2.5.

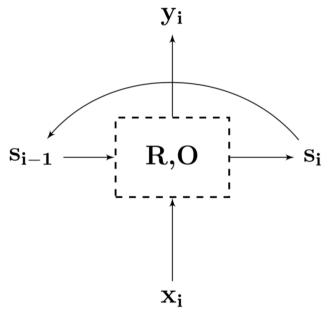


Figure 2.5: A visualization of an RNN node from Goldberg [Gol16, p. 47] [GH17, p. 165].

The visualization shows the RNN as a box containing two functions, R and O, which receives the input s_{i-1} and produces the output s_i that recursively goes back as input. The visualization could be unfolded, which means that all elements of $x_{1:n}$ would be sent to separate boxes and the output of the ith box would go to the $i+1$th box. Such a visualization would include the entire computational graph, whereas the visualization in fig. 2.5 is more succinct.

The function R that produces a state s typically has the following structure [GBC16, p. 370]:

$$R\left(s_{i-1}, x_i\right) = g_1\left(b + Ws_{i-1} + Ux_i\right) \tag{2.15}$$

where W and U are trainable weight matrices, b is a trainable bias and g_1 is an activation function.

The output function O that produces the output y typically has the following structure [GBC16, p. 370]:

$$O\left(s_i\right) = g_2\left(c + Vs_i\right) \tag{2.16}$$

where V is a trainable weight matrix, c is a trainable bias and g_2 is an *optional* activation function.

Bidirectional RNN

The output s_i of an RNN depends on the input $x_{1:i}$ meaning that it only captures information from past values. Sometimes it is beneficial to generate output based not only on the past values but the entire sequence of input data, including the future input sequence [GBC16, p. 383]. A *Bidirectional RNN* (BiRNN) generates the output y_i based on the concatenation of output vectors from a forward RNN and a backward RNN. The output of the forward RNN denoted y_i^f is based on the input $x_{1:i}$ and the output of the backward RNN denoted y_i^b is based on the input $x_{i:n}$ in reverse order. The collective output y_i of the forward and backward RNN is a combination of the forward output y_i^f and backward output y_i^b. The output y_i is:

$$y_i = C\left(y_i^f, y_i^b\right) \tag{2.17}$$

C denotes the function that combines the forward and the backward output, which most commonly is a vector concatenation [Gol16, p. 52].

Long Short-Term Memory

RNNs work well for building models based on sequence input, but training of the model is impeded by the vanishing and exploding gradients problem [GBC16, p. 390] [Gol16, p. 55]. This problem is particular to RNNs because the same weights are used for all elements of the input sequence. This means that if all elements of the sequence are the same, the gradients will vanish if the weight is less than one and explode if the weight is greater than one [GBC16, p. 391]. The longer the sequence, the worse the problem becomes. One solution has been to add skip connections so that the state s_i is not sent directly to the computation of s_{i+1}, but skips n steps so that it goes to the computation of s_{i+n} instead. This works better for some long-term dependencies but not all [GBC16, p. 395].

Another solution to the vanishing and exploding gradients problem is to use linear self-connections with a weight near one. If we have a value $\mu_t = v_t$ in step t, then a new value μ_{t+1} in step $t+1$ can be calculated with $\mu_{t+1} = \alpha\mu_t + (1-\alpha)v_t$ and if the α parameter is near one, then μ_{t+1} will be similar to μ_t thereby retaining historical information. Conversely when α is near zero, historical information is quickly discarded. Units using this kind of self-connections are referred to as *leaky units* [GBC16, p. 396].

The most effective solutions to the vanishing and exploding gradients problem in RNNs are *Long Short-Term Memory* (LSTM) and *Gated Recurrent Unit* (GRU) collectively known as *Gated RNNs*. Gated RNNs use the concept of leaky units to retain information over more steps while also being able to forget the information. In an LSTM this is done by having multiple gates as seen in the visualization in fig. 2.6.

In an LSTM, the units are connected recurrently to each other just like in the RNN, but instead of only applying an activation function between the time steps,

an LSTM has *cells* that each have a self-loop. The gates control the information flow of the cells.

The *input gate* controls whether the input goes to the self-loop state which means that the information is transferred to the next recurrence. The *forget gate* determines whether the information should be removed from the self-loop state meaning that the information cannot be used in later time steps. The *output gate* decides whether the cell should output the state information which means that the state is actually included in the computation of the time-step output instead of only being passed forward in the memory cell [GBC16, p. 399].

The number of hidden units in an LSTM refers to the number of hidden states that are passed on as memory between the time-steps. This is the same as using a vector of values as state instead of a scalar. By increasing the number of hidden units in an LSTM, the capacity of the model is increased.

Several variants of LSTMs exist. One of the variants is LSTMs with *peephole connections*. An LSTM with peephole connections use the cell state as input to the LSTM gates. This is used in most modern implementations of LSTMs [SSB14, p. 2] and can be added as a parameter in the TensorFlow implementation [Ten19x]. LSTMs with peephole connections have been found to be especially useful to find spikes with a certain time difference in the input sequence [GSS03]. These sort of spikes occur when the sound input contains a certain rhythm.

As with basic RNNs, LSTM can also be bidirectional, which is referred to as a *Bi-LSTM*.

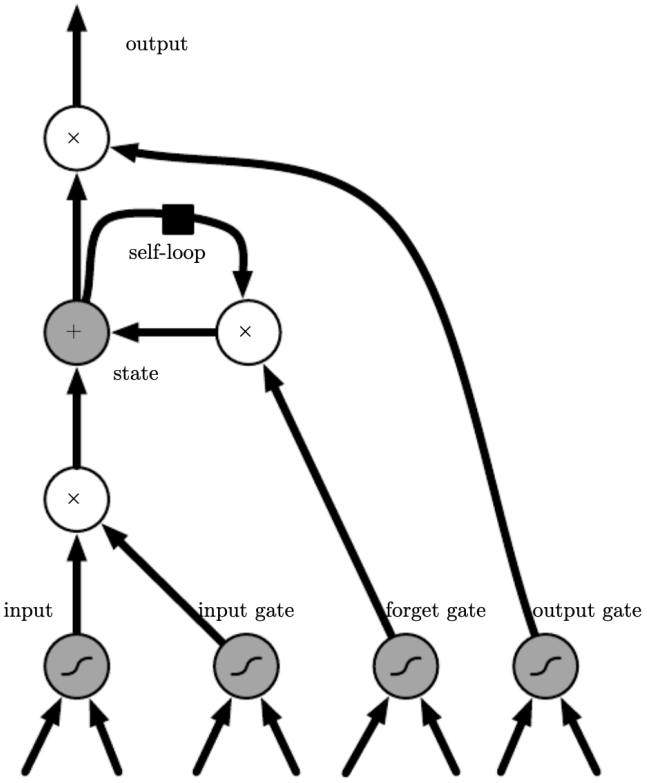


Figure 2.6: Visualization of an LSTM from Goodfellow, Bengio, and Courville [GBC16, p. 398].

4.2 Activation Functions

An *activation function* is a nonlinear function taking a scalar as input and producing a scalar as output. Because it is a nonlinear function, it is also referred to as a *non-linearity* [GBC16]. A wide variety of activation functions exist and in this subsection, the most important ones for this project are defined.

Logistic Sigmoid

A commonly used activation function is the *Logistic Sigmoid* function [GBC16, p. 65]:

$$g_\sigma(x) = \frac{1}{1 + e^{(-x)}} \tag{2.18}$$

The equation states that given the input x, the output is in the range (0,1). The function saturates with input

values that are very negative or very positive which means that the output is insensitive to changes in the input x when x diverges far from 0 [GBC16, p. 66].

Hyperbolic Tangent

The *hyperbolic tangent* function is often abbreviated as tanh and it is closely related to the logistic sigmoid function g_σ. The hyperbolic tangent function is defined as [GBC16, p. 189]:

$$\begin{aligned} g_{\tanh}(x) &= \tanh(x) \\ &= 2g_\sigma(2x) - 1 \\ &= 2\frac{1}{1+e^{(-2x)}} - 1 \end{aligned} \tag{2.19}$$

The output of $g_{\tanh}(x)$ is in the range (-1,1), but suffers from the same saturation problems as the Logistic Sigmoid [GBC16, p. 189].

Rectified Linear Units

The *Rectified Linear Units* (ReLU) activation function g_{ReLU} is [GBC16, p. 187]:

$$g_{ReLU}(x) = max\{0, x\} \tag{2.20}$$

This means that for all negative inputs x, the output is zero, and for all positive inputs x the output is x.

A variant of ReLU called *Leaky Rectified Linear Units* exists. This variant introduces a non-zero value α_i that is used when input x is less than zero [MHN13]:

$$g_L(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0 \end{cases} \tag{2.21}$$

The α value enables negative values of x to "leak" into the output with a modified scale. The typical value of α is 0.01 [GBC16, p. 188] [MHN13, p. 3].

Both the standard ReLU and leaky ReLU showed good results when used in the output layer of a neural network-based acoustic model [MHN13].

Another variant of ReLU is *clipped* ReLU, which sets a maximum value of the output [Kri10] [Amo+15, p. 4]:

$$g_C(x) = min\{max\{x, 0\}, \gamma\} \tag{2.22}$$

Here, γ is the maximum possible value of the output of $g_C(x)$, hence it is *"clipped"* at γ. This means that the range of the output is between 0 and γ. In Deep Speech 2 [Amo+15], γ is set to 20 and in TensorFlow a standard implementation with $\gamma = 6$ exists [Ten19y].

Softmax

The *Softmax* function can be used to represent the probability distribution over a finite number of classes. This makes it a bit different from the other activation functions described in this section, because the value of an element depends on the value of the other elements in the sequence. Given a sequence of n inputs $x_{0:n} = x_1, ..., x_n$, the softmax of x_i is given by [GBC16, p. 179]:

$$g_{softmax}(x_i) = \text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}} \tag{2.23}$$

All outputs of the function will be in the range (0,1) and sum to one which is why it can be seen as representing a probability distribution of n categories [GBC16, p. 178].

4.3 Back-Propagation

When input is given to a neural network model and the model produces an output, it is called *forward-propagation*. A loss-value of the output can be calculated using a loss-function to determine how far the output of the model is from the expected output. The loss value is used to update the weights of the model so that the next iteration of forward-propagation outputs values closer to the expected values. To do so, the *gradients* of the model need to be computed. The process of doing this is referred to as *back-propagation*.

If $L(x)$ is the loss of input values $x = x_0, ..., x_n$ and x_i is a value in x, then the partial derivative of L with respects to x_i is given by:

$$\frac{\partial}{\partial x_i} L(x) \tag{2.24}$$

The partial derivative measures how the loss $L(x)$ changes when only x_i changes. The gradient of the loss $L(x)$ is a vector of all partial derivatives of $L(x)$ which is denoted $\nabla_{\boldsymbol{x}} L(\boldsymbol{x})$.

When optimizing, the gradient is used to change the input x in the direction that reduces $L(x)$ most. This process is called *gradient descent*. A new input x' is found by the following:

$$x' = x - \epsilon \nabla_x L(x) \tag{2.25}$$

where ϵ is a positive scalar called the *learning rate* [GBC16, p.82].

4.4 Loss Functions

A cost function J for a model f with input x and parameters θ takes the per-element loss L to calculate the summed loss of the expectation of the data-generating distribution p_{data}:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\text{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \tag{2.26}$$

When training, only a sample of the true distribution $p_{data}(x, y)$ is taken, hence the training set distribution could be denoted as $\hat{p}_{data}(x, y)$ and the cost function J is then defined as:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\boldsymbol{x},\mathbf{y}\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] \\ &= \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right), y^{(i)}\right) \end{aligned} \tag{2.27}$$

here, the loss value is taken over m elements in the training set and the loss values are summed to represent the cost $J(\theta)$.

The loss function L used in machine learning problems depends on the problem domain. In ASR, a common method is *Connectionist Temporal Classification* (CTC) [GBC16, p. 448] [Amo+15, p. 2] [Bat+17]. CTC defines how to receive output from RNNs and other neural network constructions and convert the output to a sequence of probabilities over an alphabet A. Furthermore, CTC defines how to take the sequence of probabilities and align it to the target output sequence [Gra+06].

The probability sequence over the alphabet A is found by taking the softmax of the output from the acoustic model X generating Y. The output layer of CTC has one extra unit, which represents the blank label, hence the width of the CTC output is $|A| + 1$ where $|A|$ denotes the length of the alphabet A. By taking the softmax of this output, a value between 0 and 1 is generated for each possible label and all values sum to one, hence it represents a probability for each label.

When constructing an output, all successive identical labels are combined into a single label. The blank label between two identical labels ensures that it is possible to distinguish the case where a repeating letter is drawn out because of the speaker and the case where a word actually has a repeating letter. For instance, if the output is *to-o* where - denotes the blank label, the word could be collapsed to *too* whereas if the output is *too-* then it would be collapsed to *to*.

During training, a target label l is known. The probability of this label given the AM's output X to the CTC, denoted $p(l|X)$, is then found by summing the probabilities of different alignments of l in output Y. The loss L of the input X with target l is then calculated as:

$$L(X; l) = \sum_{(X,l)\in\mathcal{D}} -\log p(l|X) \tag{2.28}$$

where $(X, l) \in \mathcal{D}$ denotes that the sum is over all possible alignments of l.

When a model has been trained with CTC and needs to be served, the probability output Y needs to be traversed in a way that produces a relevant output without a target label l. One way of doing this is with *Beam Search*. Beam search is a modified *breadth first search* (BFS) algorithm that prune the search tree's width in accordance to a heuristic function and a search width β.

The worst-case asymptotic running time of a BFS is $O(V + E)$ [SW11, p. 541] [Shi+18, p. 11] where V is the number of vertices and E the number of edges in the graph.

The search space in this problem grows exponentially. If we define A_t as $A_t = |A| + 1$ where $|A|$ is the size of the alphabet and the one is the blank label. Then for any time step, any of the labels in A_t can be chosen. If the problem is changed into a graph then each node is the accumulated sequence of the chosen labels and each edge corresponds to selecting the next label. This means that the number of vertices in the graph grows as follows:

$$A_t^0 + A_t^1 + A_t^2 + \cdots + A_t^n = \frac{A_t^{n+1} - 1}{A_t - 1} \tag{2.29}$$

The number of edges is equal to the number of vertices except for one layer, making it $\frac{A_t^n - 1}{A_t - 1}$. Reducing this to Big O notation results in an asymptotic running time of $O(A_t^n)$. To reduce this complexity, Beam Search takes for each level of the search space only the top β values based on a heuristic function to search further through. This reduce the problem to $O(\beta n)$ [Jun06].

A weakness of beam search is that it does not guarantee to find a solution [Jun06], but in this problem all nodes in the final layer are solution nodes, hence it will always find a solution. Unlike BFS, Beam Search is not guaranteed to find the optimal solution, thus a trade-off between efficiency and effectiveness exists.

A variant of beam search is to set the beam width to one to select the best solution in each layer practically changing the algorithm from a dynamic algorithm to a greedy algorithm.

4.5 Batching

Batching, which in this project is used interchangeably with mini-batching, takes m elements from the training set and calculates the average loss of those elements. Using this technique gives a more steady training than training on single elements at a time since each $\Delta\theta$ is based on multiple training elements.

Because the elements in a batch is a random sample of elements from the entire training dataset it is a good estimate of the full training set's loss. If the estimated loss of batch $\mathcal{B}$ is denoted $J_{\mathcal{B}}(\theta)$ then the standard error of the mean of $J_{\mathcal{B}}(\theta)$ is given by:

$$\sigma_{J_{\mathcal{B}}(\theta)} = \frac{\sigma}{\sqrt{m}} \tag{2.30}$$

where σ is the standard variation of all the elements of the dataset $\mathcal{B}$ [GBC16, pp. 125-126 + 275]. As the size of the batch m grows, the standard error of the mean $\sigma_{J_{\mathcal{B}}(\theta)}$ decreases. The larger the batch size m, the less the standard error of the mean will decrease when adding one more element. This means that the effect of adding more elements to the batch decreases as the batch size grows.

Batching also impacts performance in training and serving of the model because each element in the batch can be processed simultaneously and/or grouped together to do more efficient calculations using matrix multiplication. This is especially true when using GPUs.

Starting from a simple feed-forward layer as in eq. (2.6), it is possible to batch elements in x as in the following two formulas, where the first eq. (2.31) is for an un-batched operation and the second eq. (2.32) is a batched operation with m elements:

$$\begin{aligned} \boldsymbol{h} &= g\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right) \\ &= g\left(\begin{bmatrix} W_{1,1} & \dots & W_{1,f} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ W_{v,1} & \dots & W_{v,f} & b_v \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_f \\ 1 \end{bmatrix}\right) \end{aligned} \tag{2.31}$$

$$\boldsymbol{h} = g\left(\begin{bmatrix} W_{1,1} & \dots & W_{1,f} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ W_{v,1} & \dots & W_{v,f} & b_v \end{bmatrix} \begin{bmatrix} x_{1,1} & \dots & x_{m,1} \\ \vdots & \ddots & \vdots \\ x_{1,f} & \dots & x_{m,f} \\ 1 & \dots & 1 \end{bmatrix}\right) \tag{2.32}$$

In the equations m is the batch size, f is the number of features in a given input x and v is the output size.

When choosing a batch size it is important to note that larger batch sizes make the model less generalized to unknown data [Kes+16] [GBC16, p. 276]. A consequence of smaller batches is that the learning rate usually have to be lower. The training time and number of iterations of the data would also increase when using smaller batches. Smaller batches are also harder to execute in parallel which slows down the computation further. A small batch size according to Keskar et al. [Kes+16] would be 32-512. The deficit of generalisation can be alleviated by initial training of the model using small batches that is then increased to larger batches later [Kes+16].

Batch Normalizing

Batch Normalizing was first introduced in Ioffe and Szegedy [IS15], where it is described as a normalizing technique on individual batches of data. Batch normalization allows for higher learning rates and can be used in between layers of a model to improve their independent learning of parameters.

There are many normalizing techniques. One technique takes each dimension of the data and scales to the range between 0 and 1. Another normalization strategy is to calculate the average of a dimension and then subtract each value with the average calculated. This technique is also called first moment normalization. It can be shown as:

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \widehat{x}_i &= x_i - \mu \end{aligned} \tag{2.33}$$

in which μ is the average of a dimension, x_i is the dimension's value at frame i and $\widehat{x}_i$ is the normalized value of frame i. A similar technique effecting each value by the variance is called second moment normalization. This can be defined as:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$$
$$\widehat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2}} \tag{2.34}$$

To calculate the second moment, the first moment μ is used. This is required to calculate σ^2 which is the variance of the dimension. The second moment normalization is then calculated by subtracting the first moment and dividing with $\sqrt{\sigma^2}$.

The batch normalisation from Ioffe and Szegedy [IS15] combines both the first moment and the second moment. The batch normalization has four steps as shown in algorithm 1 where x is the input values, y is the output values, m is the number of elements in the batch $\mathcal{B}$, δ is a small constant added to avoid the potential division with zero, and $\text{BN}_{\gamma,\beta}$ is the *Batch Normalizing Transform* which is scaled by γ and shifted with β and trained like an FFNN layer.

Algorithm 1 Batch normalization as described in Ioffe and Szegedy [IS15]

Calculate batch mean:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

Calculate batch variance:

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

Normalize input:

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \delta}}$$

Scale and shift the normalized values:

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$$

Hoffer, Hubara, and Soudry [HHS17] show a more advanced technique that solves the generalization problems of larger batches. The main point is that the lack of generalization comes from its less frequent updates. They suggest a technique where higher learning rates for systems with larger batches produce similar good results to smaller batches with smaller learning rates. The technique is called Ghost Batch Normalization. Instead of taking the entire batch normalization, the normalization is calculated from a subset of the batch simulating that the batches are smaller than they actually are [HHS17]. This technique is not used in this project since the batch sizes of the final model were small enough to avoid the problem.

4.6 Optimization Algorithms

An *optimization algorithm* in Deep Learning defines how to change the parameters of a model θ to reduce a loss function $J(\theta)$. The most important optimization algorithms for this project are presented in the following subsections. Many of the optimization algorithms are presented because they are fundamental for the AdaDelta optimizer, which is used in the final experiments described later in this report.

Stochastic Gradient Descent

One of the most used optimization algorithms is the *Stochastic Gradient Descent* (SGD). SGD takes a minibatch of m elements from the training dataset and takes the average of the estimated gradients of the m elements. The estimated gradients are then used to update the parameters θ with the use of a learning rate ϵ. SGD iterates until a stopping criterion is met and for each iteration a new random sample of m elements is chosen for the minibatch. It is necessary to gradually decrease the learning rate ϵ over time, hence the adapted learning rate in iteration k is denoted ϵ_k [GBC16, p. 286]. The outline of the basic SGD algorithm is seen in algorithm 2.

Algorithm 2 SGD as described in Goodfellow, Bengio, and Courville [GBC16, p. 286]

Require:
Learning rate ϵ_k
Initial parameter θ
while stopping criterion not met **do**
Sample minibatch $\{x^{(1)}, ..., x^{(m)}\}$
with corresponding targets $\{y^{(1)}, ..., y^{(m)}\}$
Compute gradient estimate:
$\hat{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
Apply update $\theta \leftarrow \theta - \epsilon_k\hat{g}$
end while

The basic SGD is improved by adding momentum to

the update of parameters θ. The purpose of doing so is to reduce the variation of direction of updates stemming from the limitation of calculating the average gradients of the minibatch instead of the entire training set. A momentum is an accumulation of past gradients in a "velocity" vector v. The velocity vector v_k is calculated based on the velocity vector in the previous iteration v_{k-1} and the current gradient estimates g. The new parameters θ are found by adding the velocity vector v. The algorithm is seen in algorithm 3.

Algorithm 3 SGD with momentum as described in Goodfellow, Bengio, and Courville [GBC16, p. 289]

Require:
Learning rate ϵ_k
Momentum parameter α
Initial parameter θ
Initial velocity v
while stopping criterion not met **do**
Sample minibatch $\{x^{(1)}, ..., x^{(m)}\}$
with corresponding targets $\{y^{(1)}, ..., y^{(m)}\}$
Compute gradient:
$g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
Compute velocity update: $v \leftarrow \alpha v - \epsilon_k g$
Apply update $\theta \leftarrow \theta + v$
end while

AdaGrad

In some cases, the loss value is very sensitive to certain parameters of the model. In those cases it is relevant to use separate learning rates for the parameters. This means that if the partial derivative of the loss of a parameter is high then the decrease of the learning rate should be fast because only small changes of the parameters are needed to produce large results in terms of the loss value. *AdaGrad* adapts the learning rate during each iteration based on the historical values of the gradient.

The pseudo code is seen in Algorithm 4 where particularly the calculation of the parameter change $\Delta\theta$ and the accumulated squared gradient r are important. The small constant δ is there to ensure that the denominator is not zero in the initial iteration. The computation of $\sqrt{r}$ is computing the L^2-norm of all previous gradients on a per-dimension basis using the Hadamard product which is denoted by $\odot$. This is equivalent to accumulating the magnitude of the gradients over time. The parameter update $\Delta\theta$ on the subsequent line is also on a per-dimension basis using the Hadamard product [Zei12, p. 2] [GBC16, p. 299].

Algorithm 4 AdaGrad as described in Goodfellow, Bengio, and Courville [GBC16, p. 299]

Require:
Learning rate ϵ
Initial parameter θ
Small constant δ
Initialize gradient accumulation variable $r = 0$
while stopping criterion not met **do**
Sample minibatch $\{x^{(1)}, ..., x^{(m)}\}$
with corresponding targets $\{y^{(1)}, ..., y^{(m)}\}$
Compute gradient:
$g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
Accumulate squared gradient: $r \leftarrow r + g \bigodot g$
Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot g$
Apply update $\theta \leftarrow \theta + \Delta\theta$
end while

Root Mean Square Propagation

AdaGrad works well for convex problems, but it has problems when applied to a nonconvex problem. This is because AdaGrad shrinks the learning rate based on the entire history which means that the learning rate could become too small before reaching a global optimum and the training could therefore be stuck in a local optimum. To prevent this from happening, *Root Mean Square Propagation* (RMSProp) changes the accumulated squared gradient formula to:

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g} \tag{2.35}$$

and the parameter update $\Delta\theta$ is changed to [GBC16, p. 300]:

$$\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g} \tag{2.36}$$

RMSProp introduces the hyperparameter ρ called *decay rate* which controls the degree of history included in the accumulation of r.

AdaDelta

Zeiler identifies in Zeiler [Zei12] the continual decay of the learning rate in AdaGrad as a problem. His solution to this problem is the algorithm called *AdaDelta* in

which it is possible for the learning rate to increase *and* decrease [Zei12, p. 3].

AdaDelta is based on two ideas. The first is the idea of accumulating r over a window which is created with an exponentially decaying average of the Hadamard product of the gradients $g \odot g$ with decay rate ρ. This is similar to RMSProp. The second idea is based on an approximation of the Hessian matrix. The Hessian matrix provides additional curvature information which is useful when optimizing a model, but the Hessian is computationally expensive to compute which is why the Hessian diagonal is approximated instead [Zei12, pp 2-3].

The pseudo code for AdaDelta is seen in algorithm 5.

Algorithm 5 AdaDelta as described in Zeiler [Zei12, p. 3]

Require:
Initial parameter θ
Small constant δ
Initialize accumulation variables $r^{(1)} = 0$ and $r^{(2)} = 0$
while stopping criterion not met **do**
Sample minibatch $\{x^{(1)}, ..., x^{(m)}\}$
with corresponding targets $\{y^{(1)}, ..., y^{(m)}\}$
Compute gradient:
$g_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), \boldsymbol{y}^{(i)}\right)$
Accumulate gradient: $r^{(1)} \leftarrow \rho r_{t-1}^{(1)} + (1 - \rho) g_t \odot g_t$
Compute update: $\Delta\theta_t = -\frac{\sqrt{r^{(2)^2} + \delta}}{\sqrt{r^{(1)} + \delta}}$
Accumulate updates: $r^{(2)} = \rho r_{t-1}^{(2)} + (1 - \rho)\Delta\theta_t^2$
Apply updates: $\theta \leftarrow \theta + \Delta\theta$
end while

Adam

Adam is an abbreviation of *adaptive moments* and is an extension of RMSProp with momentum. Adam uses momentum and is able to adapt the momentum during the training iterations. This makes the algorithm more robust to the choice of hyperparameters although the learning rate still has to be included as a hyperparameter [GBC16, p. 302].

4.7 Regularization

When training a machine learning model, a training set is used and a *training error* is calculated. This could be called an optimization problem, but what separates machine learning from pure optimization is that the machine learning model needs to be able to generalize to previously unseen data. This means that a generalization error is also found. The *generalization error*, which is also referred to as *test error*, estimates the expected error on unseen input.

A machine learning model is trying to:

1. Achieve a small training error.
2. Reduce the gap between training and test error.

If a model is not able to get a small training error, it is *underfitting* the data. If a model is not able to get a small gap between training and test error, it is called *overfitting* [GBC16, pp. 107-108]. The balance between underfitting and overfitting is controlled with the model's *capacity*. If the capacity of a model is increased, it will tend to overfit. If the capacity of the model is decreased, it will tend to underfit [GBC16, p. 109].

This subsection will describe regularization, which is intended to reduce the gap between training and test error. In Goodfellow, Bengio, and Courville [GBC16, p. 117], *regularization* is defined as:

> "*Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*"

Several regularization strategies exist. In the following subsections, the most important ones for this project are defined.

Dataset Augmentation

When the amount of training data is limited, it is difficult to generalize the model to new data. A solution to this is to generate "fake" data by augmenting the existing training data thereby getting more data with which the model is trained. When augmenting the data, it is important not to distort the data in way that changes the class which the data belongs to [GBC16, p. 233]. In the case of speech recognition, the data could be distorted by changing the recorded speech of a person saying "up" to a recording that sounds more like "top". This would degrade the quality of the model because the model would be trained to recognize "up" as "top".

In speech recognition, Jaitly and Hinton [JH13] developed a method called Vocal Tract Length Perturbation (VTLP) where spectrograms are transformed along the

frequency dimension thereby maintaining the characteristics of the words while changing the pitch of the voice saying the word [JH13].

A few data augmentations that is included in a Tensor Flow tutorial for speech commands [Ten19f] could be used on the sound extracted such as:

Padding length of the sound either before or after the sound to make the sound to listen for start and end at different points.

Offsetting starting points so that frames start at different times.

Volume manipulation changing the recorded volume by multiplying the amplitude with a value, and then clamping the result to the -1 to 1 range.

Background noise mixing the sound with other sounds, such as white noise. Here the mix can be done in many creative ways also using all the other points.

Early Stopping

The dataset used when training a model is often split into a *training* set, a *validation* set and a *test* set. The training set is used to improve the parameters of the model through loss calculation and back-propagation. The validation set can be used during training to validate how the model would respond to new input on which it has not been trained by calculating the *validation error*.

It has been observed that during training when the training error is decreasing, the validation error at some point begins to rise. This happens when the model begins overfitting the training set [GBC16, p. 239]. This means that it is an advantage to store the parameters every time the validation error improves and then restore those parameters when the model is done training. This method can be further improved by including an *early stopping* condition. With an early stopping condition, the model will stop training when the validation error has not been lowered within a specified number of training iterations. The model will then return to the parameters that resulted in the lowest validation error [GBC16, p. 240].

Dropout

Dropout provides some probability of removing non-output units from the neural network [GBC16; Sri+14, pp. 251-260]. The idea of dropout is based on ensemble methods, where several models are trained separately. This is computationally expensive which is why dropout reuses most of the model by only removing few units thereby approximating ensemble methods. The units are practically removed by multiplying their output value by zero.

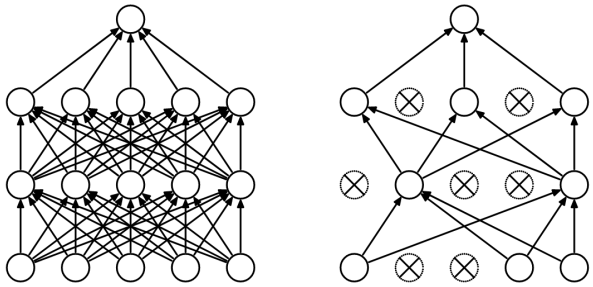


Figure 2.7: Without dropout (left) and with dropout (right) from [Sri+14].

Figure 2.7 shows the effects of applying dropout. The figure to the left has no dropout applied and the figure to the right has dropout [Sri+14]. Each iteration randomly choose units based on a *Bernoulli distribution*. A Bernoulli distribution is a discrete probability distribution of one and zero, with a probability p of getting 1 and probability $q = 1 - p$ of getting 0.

Taking the FFNN hidden layer formula from eq. (2.5) where h is the output and x is the input, applying dropout modifies the input x by dropping elements according to a Bernoulli distribution where p is the dropout probability, r_j is the jth Bernoulli random variable, r is a vector of j independent Bernoulli random variables, and $*$ is an element-wise product. Therefore the modified hidden layer with dropout becomes:

$$\begin{aligned} r_j &\sim \text{ Bernoulli } (p) \\ \widetilde{\mathbf{x}} &= \mathbf{r} * \mathbf{x} \\ h &= g\left(\mathbf{W}\widetilde{\mathbf{x}} + \mathbf{b}\right) \end{aligned} \tag{2.37}$$

5 TensorFlow

TensorFlow is an open-source platform for machine learning originally developed by Google [Mar+15]. A Tensor is an array of a variable number of axes [GBC16, p. 31] and it is one of the central data structures used in TensorFlow to represent data input and output.

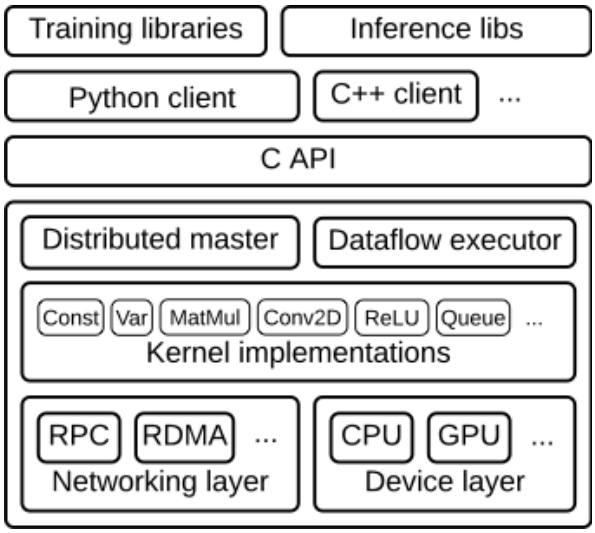


Figure 2.8: TensorFlow Architecture [Ten19i].

The computation done in TensorFlow can be thought of as a computational graph, where the nodes are the operations and the tensors are the edges between the nodes [Ten19e]. All TensorFlow programs consists of two phases:

1. Building the computational graph
2. Running the computational graph

An overview of TensorFlow's architecture is given in fig. 2.8. TensorFlow's networking layer, device layer and kernel are implemented in C and C++. It has a Python client interface and a C++ client interface which both communicate with the TensorFlow engine through a C API [Ten19i]. This means that it is possible to work with TensorFlow through the Python API while getting the performance of the C/C++ engine. Implementations in this project only used the Python client, hence optimizations within the kernel and device layer have not been performed.

When building machine learning models in TensorFlow, a computational graph is constructed with initial model parameters and the training iterations are performed by sending data through the computational graph. The parameters of the model can either be saved and loaded later in the same computational graph for further training iterations or be used for serving the model. A served model using TensorFlow Serve has a HTTP and a gRPC API.

gRPC is an abbreviation of *gRPC Remote Procedure calls* [gRP19]. It is a payload agnostic open-source Remote Procedure Call (RPC) framework that connects services across machines. This can also be called Inter-process communication (IPC).

gRPC has many features:

- low latency
- high scalability
- easy distribution
- language independent
- layered design enabling extensions
- load balancing

A basic example of gRPC usage is a client sending a request to a server that sends a response back. A gRPC system, like other RPC systems, provides a service definition Application Programming Interface (API) that can be called remotely. This service is based on protocol buffers which is a language and platform neutral marshalling mechanism for structured data [Goo19c].

gRPC provides higher-level features and consistency across multiple platforms and programming languages that HTTP libraries typically does not. Examples of these features are load balancing, failover and cascading call-cancellation [gRP19].

The high performance of gRPC comes from its parsing of data for communication, also called marshalling and unmarshalling of the data for RPC calls.

gRPC has different modes of operation [gRP19, Guides concepts] including: unary operation with a single request and response from the client and server stub, server-side streaming and client side streaming promising message ordering in both cases and bi-directional streaming with in-order messages. All modes are able to be operated synchronously, blocking and waiting for a response, and asynchronously, doing other tasks while waiting for a response.

TensorFlow has a feature called TensorBoard [Ten19g], which is used to visualize the computational graph. Different metrics can be added to the visualization, for instance the loss value of each iteration of training. In this way, TensorBoard gives a real-time overview of the training progress of the models. Another visualization in TensorBoard is the diagram of the computational graph, which can be used to compare different models.

3 Related Work

Speech recognition, along with many other deep learning problems, is very popular at the moment, with several large research groups studying and improving state-of-the-art performance. Since this area of research is very popular in recent times, because of improvements in hardware and machine learning techniques, we only mention a few of the influential papers in speech recognition and GPU learning in this chapter.

1 Baidu Deep Speech

Baidu Research - Silicon Valley AI Lab is a large research group that among other fields also do research in the domain of ASR [Han+14; Amo+15; Bat+17; Gro19]. They construct end-to-end neural networks that remove the need of advanced feature extraction techniques such as MFCC.

All of their systems use some variation of RNN networks and CTC loss. Their first publication was in 2014 and is referred to as Deep Speech 1 [Han+14]. Deep Speech 1 uses 80 log filter bank spectrogram features that are inferred in a model using a convolution layer, two FFNN layers, a single BiLSTM layer and a final FFNN layer. In the same research paper, a model using four RNN layers to handle larger amounts of data is also presented. Their models map to probability distributions of characters that use a 4-gram language model to improve the Beam Search results.

Baidu Research's second iteration of the system was published in 2015 and is referred to as Deep Speech 2 [Amo+15]. Deep Speech 2 modified and experimented with several convolution layers and up to 7 layers of LSTM and BiLSTM. The paper also presents results with and without a language model and concludes that a language model improves their performance. Furthermore, they show the effect of different strides and conclude that a stride of 2 does not decrease the WER significantly. The model is not designed for a specific language, hence it can be trained on both English and Mandarin datasets with good results. Deep Speech 2 thoroughly addresses performance issues of their implementation. They provide a wide variety of solutions, for instance a GPU-implementation of CTC loss and efficient gradient sharing between the GPUs.

Another optimization introduced in Deep Speech 2 is a batching scheduler called Batch Dispatch, which batches requests to the served model to improve throughput. The result of all these improvements is a more efficient ASR model with lower WER compared to Deep Speech 1.

In Baidu's paper "Exploring Neural Transducers for End-to-End Speech Recognition" [Bat+17] they looked at how different transducers work on the ASR systems comparing their previous model from Deep Speech 1 & 2. They call the method from Deep Speech 1 & 2 a CTC based model and compare it to an RNN-Transducer and an attention model. They conclude that both RNN-Transducers and attention models outperformed their CTC model if the models are allowed to look at the entire input meaning both forward and backward model parts. The report also says that their CTC forward-only models have better results than their forward-only RNN-transducer and attention models [Bat+17, p. 6]. In the conclusion they say that there are further work to be done in evaluating both the CTC and the attention based models. We chose not to use RNN-Transducers and attention models because of their need for the entire input for one classification and instead we work with a CTC based model.

Some of Baidu's latest research in ASR from January 2019 is a new system called *Streaming Multi-Layer Truncated Attention* (SMLTA) [Gro19]. This system does not yet have a published article but the system is described as a truncated attention model. The SMLTA model uses some form of local attention to reduce the computation time and enable efficient streaming of results.

The serving of models have also been explored by Baidu, where they research the batching of requests

to be computed on GPU [Amo+15]. The models can still be served using CPU and in SMLTA [Gro19] they find that this serving is efficient without the need of hardware acceleration through GPU.

Another feature is the "Dialect-Free Speech" that integrates Mandarin and six other Chinese dialects in a single recognition system [Gro19].

2 Mozilla Common Voice

Mozilla Common Voice is an open-source project by Mozilla that provides an open-source dataset along with the project called *Mozilla Common Voice Deep Speech* that is an open-source implementation of Deep Speech 1 [Moz18]. Mozilla Common Voice aims to collect a large amount of speech data in as many languages as possible. Usually the datasets required to train speech recognition models are owned by large companies or organisations, but this initiative makes a publicly available dataset. This dataset is the same used in this thesis and is described in further detail in chapter 4.

The open-source implementation of Deep Speech 1 [Han+14] is currently still in development on GitHub [Moz19]. It is focusing on availability and compatibility with multiple devices and provides users the ability to use a speech recognition system out-of-the-box for not only PC but mobile devices as well. The downside to this system is that the model uses the first implementation from Deep Speech 1 [Han+14], not the newer models.

3 Listen Attend Spell

Google makes ASR systems as well, one of which was proposed in Chan et al. [Cha+15] called *Listen, Attend and Spell* (LAS). LAS is an attention based RNN decoder outputting character sequences. At the time of writing the model did not outperform their previous generation CLDNN-HMM model that had a WER of 8.0% compared to the LAS model with 14.1% without language model and 10.3% with a language model.

The concept of the model is, as its name implies, to listen, attend and spell. The listener is a pyramidal BiLSTM that constructs hidden states that are used to construct an attention that using another LSTM produces grapheme characters which are probability distributions of character combinations [Cha+15, p. 3].

The pyramidal approach reduces the dimensionality of the input while also preserving long range dependencies in the sound that would be harder using convolutions with a fixed size kernel. The pyramid aspect takes two consecutive time steps from the previous layer and concatenates them into one. This halves the number of time steps of the input for the next layer of the pyramid.

The International Conference on Acoustics, Speech and Signal Processing (ICASSP) had a conference May 2019 [ICA]. At this conference the newest techniques are presented in the field of ASR and among the submissions were also a newer version of LAS [GSW19]. Guo, Sainath, and Weiss [GSW19] take the attention based model from Google further by applying a spelling correction model to correct the errors from the LAS output.

4 GPU Deep Belief Networks

Most modern neural networks are based on GPU training. The original purpose of these GPUs is graphics processing for video games, but luckily the computations required for video game graphics are efficient in the computational workload fundamental for the training of neural networks [GBC16, p. 432].

The computations for converting a game world into a 2-D visualisation on screen benefit from parallelization because each individual pixel rendered for the screen is an independent workload that can be computed in parallel. This has lead to design choices for GPUs to have large memory buffers to contain textures and models (vector meshes), high degree of parallelism to parallelize each pixel computation and high memory bandwidth to return the image results faster while having lower clock speed and branching capability than CPUs [GBC16, p. 433].

Some of the first Neural networks using GPUs from Steinkraus, Buck, and Simard [SBS05] was a connection of two fully connected layers of a deep belief network doing unsupervised learning for labeling data. This work was just before the release of CUDA enabling general purpose computing [SBS05; Var17; Coa+13] [GBC16, p. 433]. The results from Raina, Madhavan, and Ng [RMN09] a few years later using CUDA show a 70 times speedup from using GPU based computing compared to CPU based models.

Using not only one but multiple GPUs have since then been exploited [KSE12; Kri14; Coa+13]. Using multiple GPUs gives many design options of where

and what parts of the network should be located and where the computations should be done. The two most influential design choices are *model parallelism* and *data parallelism*.

Model parallelism is achieved by splitting up a model between different GPUs. One example is having the first LSTM layer on GPU one and the second layer on GPU two. This would allow each of these two layers to be double size if the GPUs are the same model, but while one GPU is working the other one is waiting. Another approach is to split the layer into multiple parts that is then distributed, for instance if the single LSTM layer is split across two GPUs. In Krizhevsky, Sutskever, and E. Hinton [KSE12] multiple network layers was split up between two GPUs, by having some of the calculations of a layer done on one GPU and the rest done on another GPU. While model parallelism allows for larger models it is often inefficient to transfer the values between the GPUs.

Coates et al. [Coa+13] takes model parallelism on GPUs to the extreme with multiple servers and measures performance of models with billions of parameters. Their experiments show that if the number of model parameters grows, it is possible to get performance gains scaling linearly with the number of GPUs, but all GPUs are not utilized 100%. Their experiments are conducted with up to 64 GPUs.

Another technique to employ is data parallelism described in Krizhevsky [Kri14] and Dean et al. [Dea+12]. Data parallelism splits work across GPUs based on individual training elements in batches. Each inference operation on an element in a batch is independent work and it scales well when distributed across GPUs.

5 Hardware Architecture

Using GPUs have given rise to many different hardware architectures. This introduces many challenges and requires specialized personnel and customized software.

Many large companies such as Amazon [Ama19], Google [Goo19a], Microsoft with Azure [Mic19] and IBM with Watson [IBM19] provide cloud platforms that give access to expensive hardware with the added benefit of no installation. Such systems are priced by the hour and in some cases by the minute with costs for instance in Google Cloud starting from 4-5$ per hour for 4 Nvidia Tesla P 100 [Goo19b].

State-of-the-art systems for machine learning employs hierarchical structures on the hardware level to take full advantage of the hardware potential. This is shown in papers such as IBM's Dünner et al. [Dün+18]. They propose a system called Snap ML that divides the system into three hierarchical levels to optimize the computation and data transfer that often is a bottleneck of both single GPU and multi GPU machine learning.

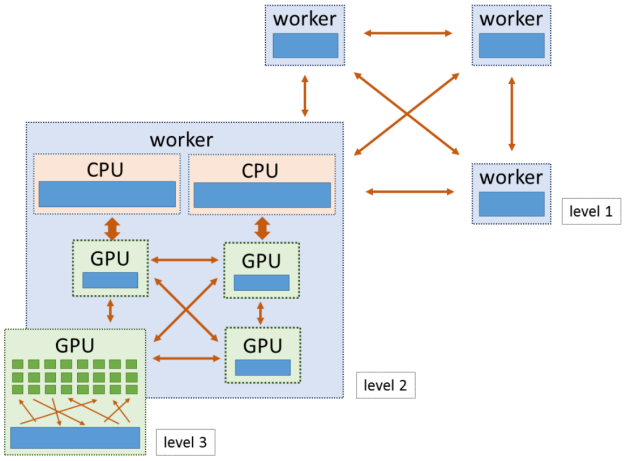


Figure 3.1: Hierarchical framework levels [Dün+18].

The hierarchical levels can be seen in fig. 3.1 and are divided into:

- Level 1: Workers in a cluster utilizing and scaling to large numbers of individual machines referred to as scaling-out. Scaling-out means adding more machines to the cluster of computers.
- Level 2: An individual machine in the cluster utilizing one or multiple GPUs referred to as scaling-up. Scaling-up is the action of upgrading a machine, in this instance it could be to add another graphics card.
- Level 3: A single GPU's environment having many CUDA and Tensor cores for massive parallelization of work.

Communication between the different levels use different protocols and physical connections, in this project Peripheral Component Interconnect Express (PCIe) [Fis19] is used. PCIe is a current expansion slot standard that is a physical connection from the motherboard to an expansion card, in this case a GPU. Nvidia provide another connection as well called NV-Link

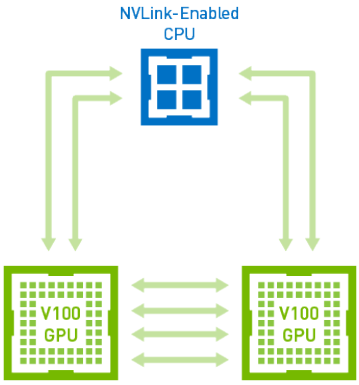


Figure 3.2: NV-Link enabled from DGX-2 [NVI17].

that provide the highest communication bandwidth between GPU and CPU at the moment [NVI17].

In fig. 3.2, NV-Link enables faster and direct communication between GPUs. Each interconnection between GPU pairs have up to 300 GB/s each with NV-Link2.

On the other hand, the PCIe based system in fig. 3.3 uses PCIe to communicate with the graphics cards, which are separated into two sets with interconnection NV-links. In this case, the PCIe limits the amount of data transferred from CPU to GPU at PCIe speeds.

Many different server providers claim that their servers offer the best support for deep learning. Nvidia themselves have servers called DGX with a slightly older system in DGX-1 using the architecture from fig. 3.3 and their newer solution with direct NV-Link as seen in fig. 3.2.

One even newer instance is IBM's AC922 [IBM18] with an open source motherboard and CPU with PCIe-4. The AC922 system can use either system architecture.

While all of these systems use state-of-the-art interconnections and boast high performance, this thesis uses a solution with a lower cost, as can be seen in table 3.1. The cost calculations of *SIM* and *RebelRig* can be seen in appendix 8. Details about the systems are in chapter 4.

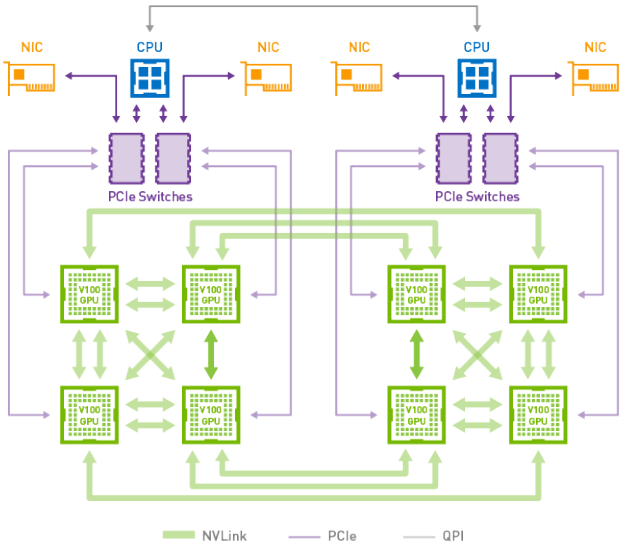


Figure 3.3: PCIe hybrid cube mesh from DGX-1 [NVI17].

Name	Cost	Cards
DGX-1	149.000 $	8 Tesla V100
DGX-2	399.000 $	16 Tesla V100
AC922	[Mor18] 166.213 $	4 Tesla V100
SIM	9.598 $	4 GTX 1080Ti
RebelRig	3.412 $	4 RTX 2070

Table 3.1: GPU rig costs.

4 Experimental Setup

1 Software Setup

Our system is split into several different modules to enable efficient training and deployment of the model after training. The basic architecture is inspired by Yu and Deng [YD15] where the model is split into Feature Extraction and Acoustic Model. The system is fitted to the TensorFlow environment, hence serving comprises a separate module in the following description. An overview of the software setup is seen in fig. 4.1 where the arrows show dataflow, the solid square boxes are modules which does not necessarily correspond to a single Python file, the dotted boxes represent Docker containers and the circles within the boxes are outlines of the content of the boxes.

The following subsections describe the basic modules of the system. First, the feature extraction is described followed by the acoustic model and finally the serving of the model. The performance benchmark is described in chapter 6.

1.1 Feature Extraction

Feature extraction (FE) has the responsibility of taking an audio file and extracting important features from the audio. The features could for instance be MFCC, magnitude spectrograms or Log Mel. The decision on which feature extraction to use is a trade-off between high quality results and efficiency of computation. If the FE results in a high number of features per audio file, the AM has more data to work with, hence it could be expected to be slower, but it also has more of the original information stored in the data file, hence it could be expected to produce results of higher quality. The trade-off between efficiency and quality is explored in the experiments described in chapter 6.

The FE takes training and validation audio files as input and saves the extracted features in files as seen in the upper left part of fig. 4.1. Since all recordings in the dataset are mono-channel with a sample rate of 16 kHz, it is loaded as such with Essentia MonoLoader [Ess19]. For each audio file, a corresponding feature file is generated.

The FE could have been an integrated part of the AM meaning that the features of an audio clip would have to be extracted for each training iteration where the audio clip is selected. The features extracted from an audio clip is deterministic, hence the consequences of such a decision would be that the same features would be extracted several times. This is unnecessary overhead, which has been removed in this project by storing the extracted features in files that are loaded by the AM later for training.

The FE is done on the CPU using TensorFlow's contribution functions for handling sound and signals [Ten19l]. The files are split into chunks of 100 files each that is then put into a queue for a thread-pool to do FE for each file. The FE has also been executed on the GPU, but this did not provide any performance advantages partly because the computation was not batched. For this reason, all experiments were conducted with FE on the CPU. As seen in fig. 4.1, the FE module is located inside the TensorFlow GPU Docker container because it is implemented to use either CPU or GPU inside the GPU Docker container.

When the sound has been loaded, it is windowed into frames of length 1024 with a step of 512 samples between each frame. The result is a frame length of 64ms with an overlap of 32ms. As mentioned, different FE techniques are used and it is calculated based on the theory described in chapter 2. Given a 10 second audio file, 312 frames of sound is generated, each frame with a dimensionality determined by the chosen FE. The dimensionality of each frame is:

512 for Magnitude spectrogram
64 for Mel & Log Mel
13 for MFCC

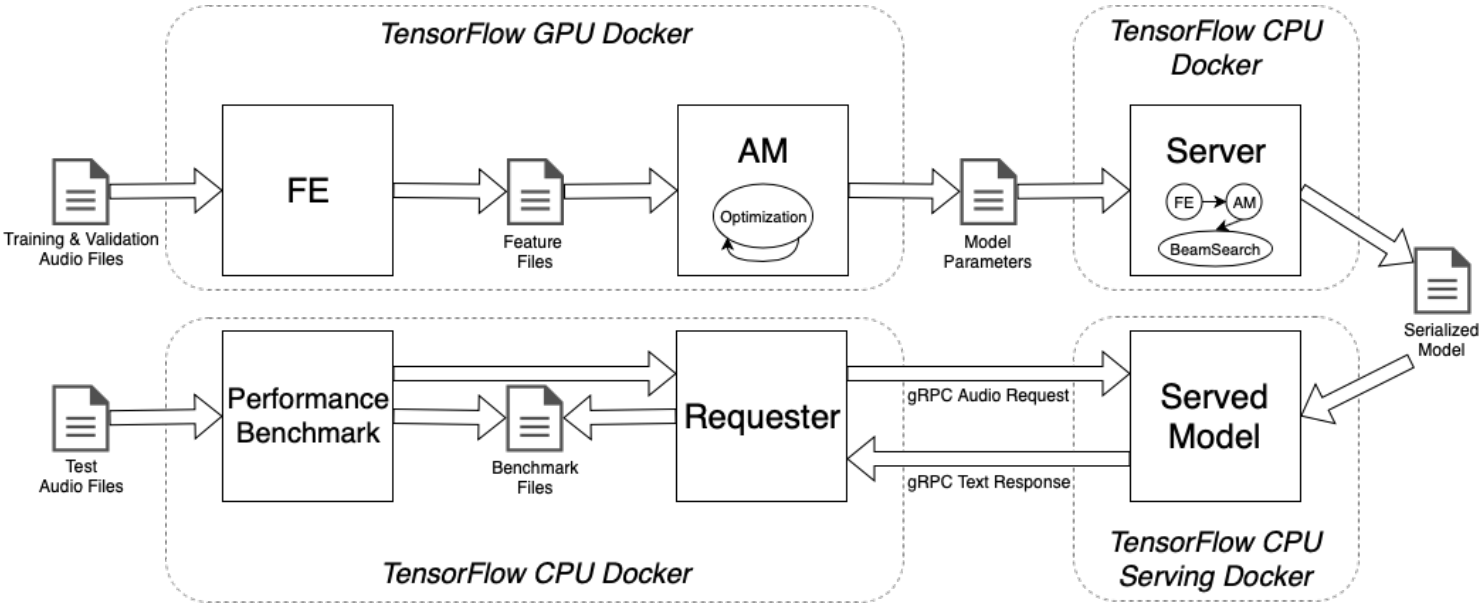


Figure 4.1: Overview of Software Setup.

Each value is represented as 32-bit Float, hence the total number of Float values for the 10 second audio file is 159744, 19968 or 3744 depending on the FE technique. This demonstrates the significant efficiency difference an FE provides.

1.2 Acoustic Model

The acoustic model (AM) has the responsibility of constructing and training the models. It loads the features from the files and iterates through them while updating the parameters of the model. Whenever the loss value of the validation set improves, the model parameters are saved in a file which can later be retrieved by the Server module to construct the final model as seen in fig. 4.1.

Optimization

Figure 4.2 provides a more detailed representation of the actions in the AM module. The optimization process of the AM in fig. 4.2 begins with the black dot and ends with the black dot with a red circle. The first action in the process is building the inference model. This action comprises setting up the layers and initializing the parameters of the model. When optimization is done on GPUs, the construction of the model is done on each GPU included in the system. This is described in further detail in section 3. The program has been

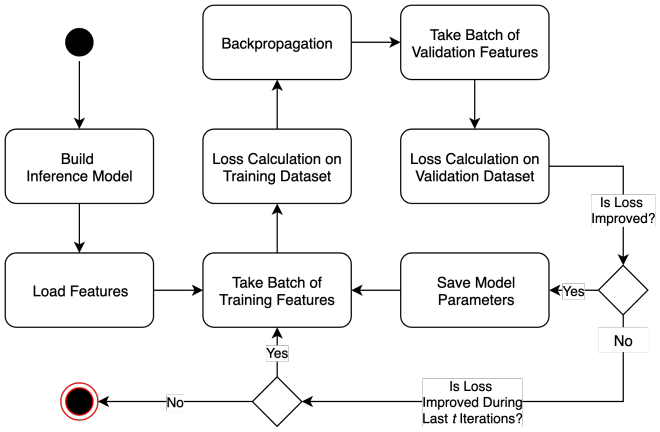


Figure 4.2: Optimization in AM based on training and validation dataset.

set up to choose between several different inference models. Each inference model is given a number and the program takes a number as a parameter to pick a specific inference model. This makes it possible to switch between models in the different experiments. The models are referred to as *M<number>* in this report, hence model number 33 is called M33. An overview of the different models is found in appendix B. Several models seem very similar, but small differences in the TensorFlow-implementation or parameter settings mean that they are presented as different models. Indexing the models also enables the Server module to load the parameters with an original model. If the old models are removed from the code, the model could not be loaded again by the Server module even if the model parameters are still stored. In this way, the indexing of the models enables backward compatibility. The weights in the model are initialized using Xavier Initialization as described in chapter 2 section 4.1. This is performed with *tf.contrib.layers.xavier_initializer* [Ten19q]. The Xavier Initializer was chosen based on small experiments conducted in initial stages of the project. Different initializers available in TensorFlow were tried and the Xavier Initializer demonstrated faster loss convergence, hence it was chosen for the rest of the project.

When the inference model is constructed, the features are loaded from the files. The features of the training dataset and the validation set are both loaded. The amount of RAM available in both servers used in this project are great enough to load both datasets in the beginning and keep them in memory throughout the optimization. Both feature sets are divided into equally-sized shards. One shard is created for each GPU available when training on GPUs. This means that each GPU has a unique part of the dataset on which it optimizes the parameters of the model.

When the inference model is constructed and the features are loaded in shards, the optimization of the model can begin as described in chapter 2 section 4.6. Initially, four different optimization algorithms were tried out. All optimization algorithms require a specified learning rate, hence the program has been implemented to accept the learning rate as a parameter. This makes it easier to try different learning rates and determine which learning rate works best.

After initial tests using an SGD optimizer, we switched to an AdaGrad optimizer from TensorFlow *tf.train.AdagradOptimizer* [Ten19n]. The AdaGrad optimizer is able to adapt the learning rate as specified in chapter 2 section 4.6, but when the loss-value decreased to a certain level it could not decrease further. This corresponds to the description in chapter 2 section 4.6 where it is described that the learning rate becomes too small to get to the global optimum. Instead, the training iteration is stuck in a local optimum which is seen in our case by having a medium-high loss-value that is not improved through the iterations.

To mitigate this problem, the Adam optimizer in *tf.train.AdamOptimizer* [Ten19o] and the AdaDelta optimizer in *tf.train.AdadeltaOptimizer* [Ten19m] were tried out. The final decision was to use the AdaDelta optimizer, which showed good results in our initial experiments and is also used in Bahdanau et al. [Bah+16]. The program takes a number specifying the optimization algorithm as an argument. This means that it is easy to switch between the different optimizers, although all experiments described later in this report is based on AdaDelta.

When the optimization algorithm is decided upon, the next action from fig. 4.2 to describe is the batching of training features. The size of the batch is also an argument given to the program. The performance effects of different batch sizes are determined by the experiments described in chapter 5.

The next step is calculating the loss of the batch. A CTC loss calculation as described in chapter 2 section 4.4 is performed for each element in a batch with *tf.nn.ctc_loss* [Ten19k]. The mean of the losses is computed and based on this the gradients are calculated. When doing this on GPUs, the gradients from all GPUs have to be collected and averaged as well. This is described in further detail in section 3.3.

In fig. 4.2, the step after back-propagation is taking a batch of validation features. The loss value of the validation batch is calculated on the same version of the model except any dropout is removed from the model. The purpose of calculating the loss value of the validation set is to test the generalization capabilities of the model during training iteration. The validation set loss value is used as a stopping condition for the training and it ensures that the model parameters that are expected to work best on the test dataset are always stored. This is represented by the next actions in fig. 4.2. If loss is improved during the iteration then the model parameters are stored. If the loss is not improved and has not been improved during the last t iterations, where t is an integer specified as a parameter to the program, then

the training of the model is terminated. This design decision adheres to the theory described in chapter 2 section 4.7.

If the training has not been terminated, the training continues iterating by taking a batch of training features again.

Inference Model

Using TensorBoard as mentioned in chapter 2 section 5, the models defined in appendix B can be inspected further. As an instance of a model, the computational graph of a single GPU Tower of M33 is presented in appendix D. It demonstrates the different layers that a model is composed of and the flow of data through the layers. The figure shows two independent computational graphs. The left graph is the computation on the training dataset and the right graph is the computation on the validation dataset. The figure is read from the bottom, where an *Iterator* node is seen. The input features flows to the *mfcc_norm* node and the target sentences and target length flows directly to the *CTCLoss* node. The *mfcc_norm* node is a batch normalization on the feature input as described in chapter 2 section 4.5 performed in TensorFlow with *tf.layers.batch_normalization* [Ten19s].

After the initial batch normalization, the data flows to the convolution layer *conv1d* which is a one-dimensional convolution using *tf.nn.conv1d*. This convolution takes different types of padding and different strides as arguments. In the case of M33 the padding is set to *valid* meaning that there is no padding, but most of the models use *same* padding meaning that the input dimensionality equals the output dimensionality. Different stride settings has been experimented with in chapter 6. Another argument for conv1d is whether to use NVIDIA's *CUDA Deep Neural Network library* (cuDNN) which is an optimized library for neural networks on GPUs. This has been activated in our project. The convolution layer is followed by a ReLU node and a normalization layer. This could also have been represented in the same node as conv1d, but in this specific case the nodes have been expanded. The instance of M33 in appendix D does not include a dropout layer, but this kind of layer could also be represented as an individual node in the graph after mfcc_norm.

The next five layers consist of LSTMs, which also includes activation functions, batch normalizations and dropouts between the layers although it for the sake of brevity has not been displayed in the graph. M33 has five LSTM layers, but the number of layers is different for some of the other inference models and the effect of changing the number of LSTM layers is reported in chapter 6. The LSTM is implemented with tf.contrib.rnn.LSTMBlockCell [Ten19r] and tf.nn.dynamic_rnn [Ten19w]. The bi-directional LSTM makes use of tf.nn.bidirectional_dynamic_rnn [Ten19u] instead of tf.nn.dynamic_rnn because it is able to take both a forward RNN cell and a backward RNN cell. The LSTMBlockCell can be initialized with or without peephole connections as described in chapter 2 section 4.1. The effect of adding peephole connections was explored during the project and it is described in chapter 6.

A cuDNN version of the LSTM cell also exists. This LSTM was approximately 30% faster during the training iterations on our models, but the model parameters saved in the end could only be used on GPUs and generally requires a different handling during the final serving preparation and serving of the model. Because of this, only the standard LSTMBlockCell was used during the final experiments in this project.

The last part of M33 before loss calculation is the *logits* node which is the fully-connected output layer. It is implemented with *tf.layers.dense* [Ten19t] and the output size is defined to be the size of the alphabet + 1 as defined in chapter 2 section 4.4. It is followed by a ReLU-activation which is either a standard ReLU, as in M33, or a clipped ReLu with $\gamma = 6$ as defined in chapter 2 section 4.2.

The program takes several arguments which define the model, the training procedure and the information stored and printed by the program. The arguments to the program are defined in appendix E.

1.3 Serving

Serving the ASR model comprises two steps as seen in fig. 4.1. First, the model parameters have to be loaded into the computational graph to save the serialized model. Then the serialized model has to be served using a TensorFlow Serving Docker image [Ten19j].

The Server module is executed in a TensorFlow CPU Docker container. This is in contrast to the FE and AM modules, which were executed in a TensorFlow GPU Docker container which includes the Nvidia CUDA Toolkit and other toolkits needed for GPU execution [Ten19d]. The Server module loads

the model parameters from file, constructs the FE computational graph, constructs the AM inference model, constructs a Beam Search computational graph, combines the three computational graphs and applies the model parameters loaded from file to the combined computational graph. The construction of FE and AM in the Server module reuses code from the other modules, but the Beam Search is only used in the Server. The Beam Search is implemented with TensorFlow's *tf.nn.ctc_beam_search_decoder* [Ten19v] with a preceding SoftMax of the logits output from the AM inference. The Beam Search is set to only select the top path and it takes the beam width as an argument. The width argument is specified when executing the Server module, hence experiments can easily be conducted with different widths. The final computational graph as a combination of FE, AM and Beam Search is saved as a Protocol Buffer file (.PB), which is a mechanism for serializing structured data [Goo19c].

The arguments to the Server module is:

Run Test: This is a Boolean argument that if set to true will construct the computational graph and immediately give input to and print output of the combined model without saving the model as a PB-file. In this way, a model can quickly be used without saving and loading the combined model first.

Model: Specifies the path to the saved model parameters. This path is also used to load the right inference model, find the network size, the convolution stride and other model information relevant when building the final computational graph.

Beam Width: The beam width of the Beam Search.

As mentioned in the beginning of this section, the model is finally served using the TensorFlow Serving Docker image. The path to the PB-file storing the model is mounted when initializing the Docker container and the model is served with a gRPC and HTTP interface on the ports specified. The served model accepts HTTP and gRPC requests as long as the Docker container is running. If a new PB-file is stored in the mounted path, TensorFlow Serving is able to load the new model without server downtime and serve both the new and the old model concurrently [Ten19b]. This means that clients can still use the old model or begin using the new model. The CPU Serving Docker image is chosen in this project, but a GPU Serving image also exists [Ten19j]. Serving on GPUs would be an advantage if batching is applied during serving. Batching during serving has the potential to improve throughput, but it could also increase latency of individual requests as reported in Deep Speech 2 [Amo+15]. The increased latency occurs because the incoming requests have to be collected and batched. The higher throughput occurs because resources are utilized better, especially if the Docker GPU Serving is used. Serving the model on GPUs is discussed further in chapter 7.

2 Workload Setup

The data used in this project is the Mozilla Common Voice dataset for English [Moz18], which is a dataset constructed in the project Mozilla Common Voice briefly mentioned in chapter 3.

Mozilla Common Voice is an open source project containing speech from people all over the world. The data is collected from people reading predefined sentences that are validated by other users. The dataset contains 541 hours of validated English sentences and the dataset is still growing. The version used in this report is from February 8, 2019. The dataset provides a split along two dimensions. The first dimension indicates whether a speech recording has been validated by other users and the second dimension indicates the purpose of the dataset.

The first dimension splits the dataset into *invalid*, *other* and *valid*. Valid contains clips that have at least two listeners verify that the sound and the text correspond. Invalid has at least two listeners who says the sound and the text does not correspond. Other contains the sound with no votes or an equal number of valid and invalid votes. Only the data validated by other users has been used in this project.

The second dimension of the dataset provides a split of the data into *test*, *train* and *validation* sets. The validation set is referred to as the *dev* set in Mozilla Common Voice so that it is not confused with the *valid* part of the dataset, which refers to the part of the dataset which has been validated by other users. The fixed split into three parts enables easy comparison with other implementations using the same datasets if one remembers to use: train set for training the model, validation/dev set to parameter tune and verify the generalisation of the model, and test set for testing the error rate on a finished model.

Distributions of the dataset along with size and number of unique words is shown in table 4.1.

In chapter 2 section 2, the speech properties are described with *utterance*, *speaker model* and *vocabulary*.

The *utterance* of Mozilla Common Voice is continuous speech, since it consists of recordings of people reading pieces of text in a natural way. The utterances do not reach the level of spontaneous speech because the users are reading a text and it is not at the level of connected words because the users do not pause between any of the words in a given sentence.

The *speaker model* of Mozilla Common Voice is speaker independent, because the dataset contains speech from a great number of different speakers from different parts of the world. This poses a challenge to the ASR model, because it needs to generalize to different voices and accents.

Mozilla Common Voice contains 10721 unique words in total. The number of unique words in the validated training dataset is 8004. A Large-Vocabulary Continuous Speech Recognition (LVCSR) dataset generally have roughly 20,000 to 60,000 unique words according to Jurafsky and Martin [JM09], hence the vocabulary of Mozilla Common Voice could be said to be medium sized. If the output of the ASR model is character-based, it would be able to output out-of-vocabulary words. By contrast, if the output of the ASR model is word-based it is only able to output the 8004 unique words in the training set. Only seven words in the test dataset and ten words in the validation/dev dataset are not represented in the training dataset. These words are out-of-vocabulary. The character-based model is able to output the out-of-vocabulary words, but the word-based model is not able to do so.

Another important observation about the data is that there are duplicate sentences in the training, dev and test dataset. For instance, the sentence *"you got your sea legs yet"* is represented nine times in the training, one time in the dev and two times in the test validated dataset, but all the utterances are different. This makes it easier for the model to identify the sentences in the test dataset because it is able to train on similar sentences in the training dataset.

As mentioned in the beginning of this section, the Mozilla Common Voice dataset contains 541 hours of speech. This is appropriate for a master thesis, but it is a small dataset compared to the data used in Deep Speech 2 [Amo+15, p. 15]. They use 11940 hours of English speech to train their model, while also using data augmentation to further bolster their dataset. Google's Listen Attend Spell (LAS) system was trained on 2000 hours of raw data, that was augmented to 40000 hours using various techniques of data augmentation [Cha+15, p. 6].

Folder Name	Samples	Size	Unique Words
cv-invalid	25.403	1.2G	6.147
cv-other-dev	3.022	103M	4.066
cv-other-test	2.961	99M	4.042
cv-other-train	145.135	4.8G	10.616
cv-valid-dev	4.076	147M	3.523
cv-valid-test	3.995	145M	3.505
cv-valid-train	195.776	6.9G	8.004
Total	380.368	14G	10.721

Table 4.1: Datasets in Mozilla Common Voice [Moz18].

2.1 Data Cleaning & Preprocessing

All data in Mozilla Common Voice is lowercase with no alphanumerics and punctuation except space and apostrophe. This is a common practice where some have small variations including comma and period, for instance like in LAS Chan et al. [Cha+15, p. 6].

The dataset contains varying lengths of audio files. This is a problem when batching data together because each element in a batch must have the same length in TensorFlow. This forces each element in a batch to be padded to at least the largest element in that batch.

The model is trained on all elements in the dataset that are under a specified length and padding all other elements up to the same length to achieve consistency in the length of inputs to our model. Before padding elements, the actual length of the elements are stored enabling the calculation of loss on parts containing sound while ignoring the padded values. It is acceptable to do cutoff and padding in our case since each file contains a single sentence of approximately same length. This is unlike other speech systems such as Deep Speech [Amo+15] that contain both single word, sentences and conversations making the length of the files more diverse. Our model is still able to handle dynamic size input after the training and files longer than our training's cutoff length.

After reducing the data based on a cutoff length it is further reduced by setting the stride and convolution width of the models as specified in eq. (2.12). This reduces the amount of output features of the model, but since the CTC loss requires the output to have at least

stride	Training size
4	19.350
3	61.576
2	164.453

Table 4.2: Effects of stride on character-based models.

the amount of input features that is used for all output characters or words, we filter the files further according to the criteria in eq. (4.1). W is the number of input frames before padding, T_{size} is the number of words or characters in the target sentence, F is the receptive field size and S is the stride:

$$T_{size} \leq \frac{W - F}{S} + 1 \tag{4.1}$$

The equation is almost the same as eq. (2.12), but without the padding variable since only valid convolutions are applied in our newest models. T_{size} must be less than or equal to the number of output features which corresponds to the right hand side of the equation. If T_{size} was higher, it would not be possible to output the entire correct sequence because the length of the output sequence would always be less than the length of the target sequence T_{size}.

Table 4.2 shows the reduction of the training data if the cutoff length is 300, convolution width is 8 and the stride is set to the different values. The number of elements is reduced from a total of 195.776 in the training dataset. In the table it is seen that the number of training elements decreases significantly when the stride is increased. By increasing the number of frames generated from an audio clip during feature extraction as defined in chapter 2 section 3.1, a lower number of elements would need to be discarded. Increasing the number of frames corresponds to increasing W in eq. (4.1), hence the right-hand side of the inequality is increased.

3 Hardware Setup

During this project two setups have been used to train and serve the models. We set up both systems from hardware to fully functional systems. The systems are not as powerful as other GPU clusters used for speech recognition such as the one used by Baidu in Deep Speech 2 [Amo+15] using a setup with 8 or up to 16 Nvidia Tesla V100s.

3.1 System 1 - RebelRig

The first system, from here on called **RebelRig**, is a low cost GPU system designed to minimize the cost of creating a multi-GPU system. It is set up with 4 Nvidia RTX 2070 at 1.7GHz, an Intel i7 6700k processor having 4 cores hyper-threading to 8 logical cores at 4 GHz, and a low cost mining rig motherboard ASRock H110 Pro BTC+ [AsR19]. The downside to this system is that the motherboard's communication with the graphics cards is over PCIe 2.0 x1 except for the first card that is PCIe 3.0 x16. Another downside is its limited 16GB DDR4 Dual Channel RAM at 2133 MHz. The OS and storage is on a Micron M600 512GB SATA SSD.

3.2 System 2 - Sim

The second system, from here on called **Sim**, is a dual socket Super-Micro server with a X9DRG-QF motherboard [Sup18] using two Intel XEON E5-2630V2 each running 6 cores with hyper-threading resulting in 24 logical cores at 2.6 GHz using 126GB DDR3 Quad channel RAM at 1600 MHz. On top of this, the system is using 4 Nvidia GTX 1080 Ti each at 1.5GHz. The OS and storage is just like the RebelRig system on a Micron M600 512GB SATA SSD. The setup's architecture can be seen in appendix C.

3.3 GPU DNN Training

The training of our models are in the case of both systems on multiple GPUs. Training on GPUs enables a high level of parallelism and enables many times the performance of CPU-based training. This gives some challenges in the distribution and collection of data, and because we use TensorFlow we have to manage most of this ourselves allocating different parts of the computation to a specific graphic card and CPU [Ten19p]. Most frameworks for deep learning support the distribution of computations to multiple GPUs, for instance: Pytorch [PyT], Keras [Ros17], and Caffe [Jia+14].

While TensorFlow gives the ability to specify where a computation should go, it does not per default specify where the variables are located. To enable quick updates we want to locate the updating and management of variables on the CPU while having the main bulk of computations made on the GPUs. To do this we use tower abstractions, identifying each graphic card as an isolated environment as in fig. 4.3 [Ten19h].

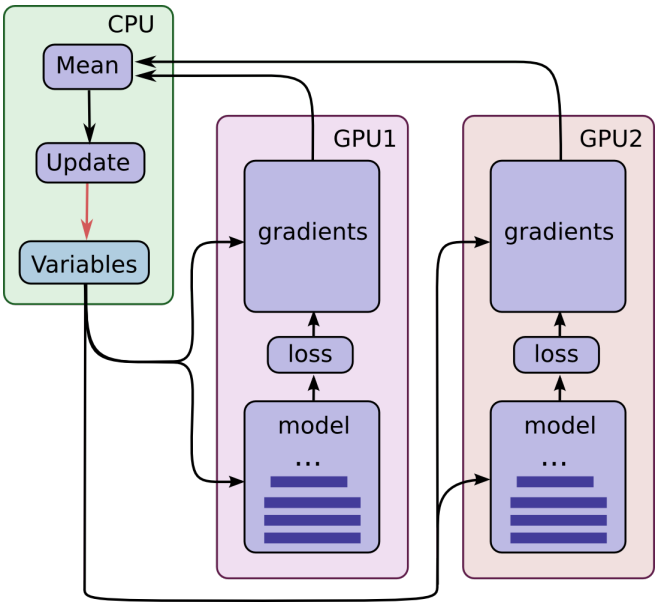


Figure 4.3: Visualization of multi-GPU setup from TensorFlow [Ten19h].

The variables located on the CPU in fig. 4.3 are the model's parameters. They contain all the weights for the different model layers. For each execution loop these are transferred to each of the towers for calculation of loss and gradients. The loss and gradients are then calculated and returned to the CPU which calculates the mean gradient for all GPUs. The gradients are used to update the weights in the variables and the loop repeats for the next iteration.

While fig. 4.3 shows how to allocate the model, it is missing the data. To help with this, we use the data input pipeline from TensorFlow [Ten19c]. This data input pipeline is shown in fig. 4.4 and enables the CPU to prepare the next batch of data (the blue colour) while the GPU is calculating the result of the previous batch (the green colour). This significantly reduces the time the system spends idle (the red colour). This is a non-blocking preparation allowing the CPU to prepare more than one batch of data for the GPU filling up a defined size buffer.

The batches of input data are shuffled for a more even training of the network. The input pipeline also allows for data augmentation increasing the amount of potential data but this functionality is not used in this project.

In systems with more than one GPU, more data input pipelines are prepared isolated in their own threads. In this report the dataset is divided into fractions, one for each GPU, and used for different input pipelines. This gives a performance that scales near linear with the number of GPUs as shown in chapter 5.

3.4 Mixed Position Training

Another optimization that can be made at the cost of a slightly lower precision is mixed position training [NVI19b; Mic+17]. Mixed position training is training the model using lower floating point precisions while maintaining a master set encoded in higher precision.

Algorithm 6 From NVIDIA [NVI19b]

$W_{32} \leftarrow Weights$
$S \leftarrow LargeValue$
for each training iteration **do**
  $W_{16} \leftarrow W_{32}.toFloat16$
  $I \leftarrow InputQue.Deque()$
  $loss \leftarrow FP(W_{16}, I)$
  $scaledLoss \leftarrow loss \times S$
  $gradients \leftarrow BwPropergate(scaledLoss, W_{16})$
  if $gradients.contains(Inf||NaN)$ **then**
    $S \leftarrow reduce(S)$
  else
    $gradients \leftarrow gradients \times 1/S$
    $W_{32} \leftarrow WeightUpdate(gradients, W_{16})$
    if no Inf or NaN in last N iterations **then**
      $S \leftarrow increase(S)$
    end if
  end if
end for

While using mixed position training does produce higher speeds, it also has some deficits. The main one is loss of precision and vulnerability to vanishing and exploding gradients. A solution to these problems is shown in algorithm 6. In the algorithm the loss is scaled higher to preserve smaller gradient values thereby removing the vanishing gradient problem. Training iterations that contain Infinite or Nan values are ignored and instead the loss scaling is reduced, which solves the problem of exploding gradients.

Some of the first to employ this technique during training are the Baidu research team. Deep Speech 2 [Amo+15] describes very briefly its use in the served model but the actual details are further described in Micikevicius et al. [Mic+17] that show increases in performance using the lower 16-bit precision.

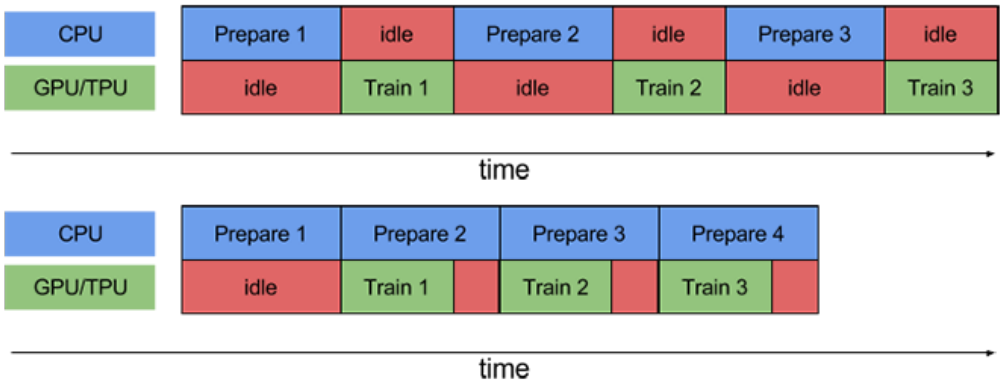


Figure 4.4: Pipeline without preprocessing (top) and pipeline with preprocessing (bottom) from TensorFlow [Ten19c].

The performance improves with regards to computational speed, transfer speed and memory usage. The computation speed is increased because the GPUs can execute twice the amount of 16-bit operations than 32-bit in a single tick [NVI19b]. This is much like *Single Instruction Multiple Data* (SIMD) commands on the CPU. The data transfer is lowered since each value is reduced to half the size. This improves performance especially in systems where data transfer is a bottleneck which is the case in this project. The memory usage of the GPU is also lowered enabling larger models on the GPUs. This has a large impact on the ability to use smaller machines for training and serving.

Overall mixed position training could produce up to a 3x speedup on "the most arithmetically intense model architectures" [NVI19b] or according to Micikevicius et al. [Mic+17] 2-6x speedup.

An independent benchmark by Perbos-Crinck [Per19] shows less improvement in performance using 16-bit precision. It shows that graphic cards such as the 1080Ti used in Sim does not benefit from using 16-bit floats and actually performs worse using 16-bit floats. On the other hand, the performance of RTX 2060 increases when changing from 32-bit to 16-bit because 16-bit precision makes it possible to double the batch size with the same memory usage. The observation from this article is that the newer generation of graphics cards benefit more from 16-bit.

The reason behind this difference between the generations is the micro architectures difference going from Pascal [NVI16] on the GTX to Turing [NVI18] on the RTX. RTX's Tensor-cores are able to utilize the mixed precision [NVI19a] while the GTX does not have such functionality on its hardware.

Another suggestion from NVIDIA [NVI19b] is that reduction layers such as mean, variance, SoftMax and batch normalization should not be reduced in precision while training. This is further enforced by the implementations in TensorFlow that in many of the predefined functions require at least 32-bit encoding, such as the implementation of batch normalization in tf.layers.batch_normalization [Ten19s]. The suggestion from NVIDIA [NVI19b] requires the models to switch between the 16 and 32 bit encoding constantly between each layer since our models contain a batch normalization between each layer. This introduces overhead in the casting of the values back and forth and ultimately reduces the performance. Mixed position training has been tried during this project with M25, but the performance was not improved. For this reason, the other implementations of this project are not making use of mixed position training.

5 GPU Training Performance

When training on more than one GPU, it is important to be aware of the neural network topology, since different topologies have large impact on computation time. Furthermore, the trade-off between model accuracy and efficiency has to be handled and potential performance bottlenecks identified.

To examine the above-mentioned issues, we have run multiple benchmarks using our models to determine which parameters give an efficient training. Efficient in this context is a high throughput of samples in the training phase while also producing quality results in the served model.

One way to measure and reason about the system is the USE method that can be summarized as: *"For every resource, check utilization, saturation and errors"* [Gre13, pp. 181- 184] [Gre17]. Resources refer to the specific physical components such as CPU, disk, busses, and GPUs. The resources we look at in this report is limited to CPU, busses, and GPU. The busses specifically refer to the PCIe busses used for the communication with the GPUs.

Utilization specifies how much time the resources spend servicing work. *Saturation* defines the amount of extra work for the resource that cannot be serviced immediately and is therefore queued. *Errors* refer to events that produce errors in the running system, hence it is work which the resource could not complete.

The metric for utilization in this project is volatile GPU utilization, which is *"percent of time over the past during which one or more kernels was executing on the GPU"* [NVI12, p. 90].

In the context of training a neural network, saturation becomes harder to reason about since often infinite amounts of data is queued for processing. The saturation of the training could focus instead on the memory transport and data pipelining trying to avoid memory bottlenecks thereby making the computations compute-bound. We consider the system saturated if the GPUs are compute-bound while being compute-bound it would also be considered fully utilized.

Memory-bound is opposite to compute-bound, and is a distinction between when the GPUs are internally using their memory bandwidth efficiently [NVI19b]. This terminology can be used for CPUs as well and is comparable to CPUs' state of cache where a CPU is compute-bound when the cache is hot and memory-bound when the cache is cold [Gre13, p 32]. In the context of this project, the term memory-bound is used to describe when the system is memory-bound on the outside connection to the GPUs, in this case the PCIe bus connection from CPU to the GPU.

In addition to the metrics in the USE method, throughput is an important metric. Throughput is volume processed over time [Gre13, p. 27], hence in the context of training a machine learning model it can be defined as the number of trained samples per second.

Other metrics for throughput could have been used such as *floating point operations per second* (FLOPS) [NVI19b]. But generalizing this to all the models and operations would add little to no value to the experiments. Another option is *Frames processed per second* used in Strom [Str15] when training the Alexa speech recognition system. This is an appropriate measurement, and can be calculated from our values by multiplying by 300 since each elements have 300 frames while we train.

A reason why we use samples per second is that it gives a better indication of the performance to be expected from a final served model from a user perspective if using GPUs.

Some guidelines from Nvidia that should further improve performance is followed where appropriate [NVI19b], such as: layers should be wide rather than many deep layers; batches, layers and convolutions should be multiples of 8; and sequence problems should be padded to a multiple of 8. This is to follow the optimal tiling of the operations for the GPUs to be run on Tensor cores and secondly on CUDA cores.

Another connection that could be measured is the connection from CPU to disk and CPU to RAM. In the system implemented, all training data is loaded into RAM once using a thread pool to parse it for a training session, but since the data loading from disk to RAM is a one time operation it is not further examined.

Further research of the performance implications of dual/multi socket CPU's are possible because of their separated memory lanes and separated connections to the graphics cards. This type of experimentation is possible on the SIM setup as seen in appendix C. Since we did not include code to pin processes to specific cores, it introduces a high chance of the CPU to communicate with graphics cards not in direct connection requiring the CPU to send the data from its memory through the other CPU and on to the graphics cards introducing high overhead. Instead of this we focus on the scaling performance of using multiple GPUs.

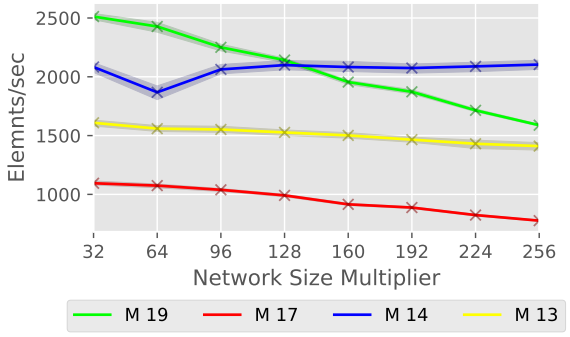
1 Single GPU

To measure the performance of the system, we start with measurements on a single GPU to get information about what works well without the added overhead of multiple GPUs.

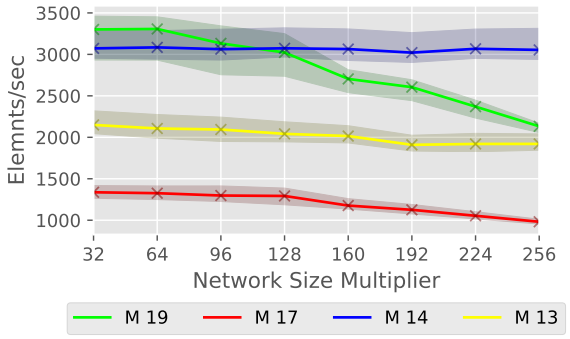
Two different experiments are conducted on RebelRig. One experiment is conducted on the RTX 2070 GPU connected with PCIe 2x1 and the other experiment is conducted on the identical GPU connected with PCIe 3x16. Four different models are trained through 250 iterations with different model sizes. The number of elements per second is noted for each iteration.

The results are seen in fig. 5.1. Figure 5.1a shows the PCIe2x1 performance, fig. 5.1b shows the PCIe3x16 performance and fig. 5.1c shows the difference between the performance of the two. The coloured lines are the median values, and the marked areas around the lines are the first and third quartile. The x axis is the network size multiplier which influences the different LSTM layers so that they each are the size specified in the multiplier. Convolutions are not changed by the network multiplier.

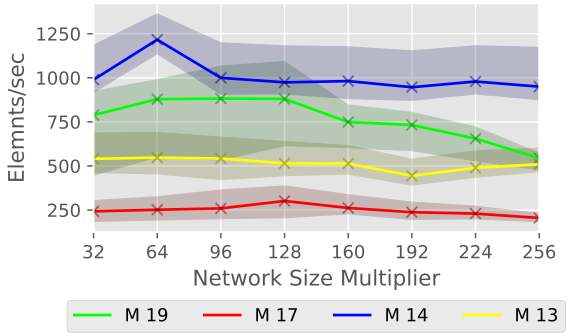
M13 consists of a single LSTM layer and a fully-connected output layer. M14 only has a fully-connected output layer. M17 consists of three LSTM layers with batch normalization and a fully-connected output layer. M19 consists of a batch normalization before a convolution layer with stride 2, width 8 and kernel size 256



(a) PCIe2x1 Performance.



(b) PCIe3x16 Performance.



(c) Performance Difference.

Figure 5.1: Network size training throughput experiment.

followed by three LSTM layers and a fully-connected output layer.

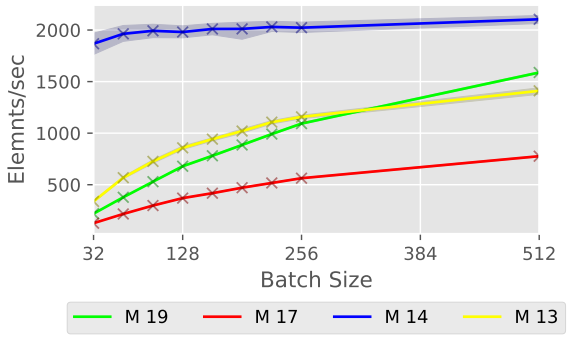
M14, the blue line, has the same performance for all network size multipliers since it does not have an LSTM layer. The performance drop at multiplier size 64 is due to other processes running while that particular benchmark was run. M13 and M17 are affected slightly by the network size growing, with larger networks giving lower throughput. M19 starts with a performance that is higher than the simple feed forward M14. This is because of the stride included in M19 which halves the amount of values that is sent through the LSTM. All of the above is the case for both the PCIe2x1 and PCIe3x16 connected GPUs.

As seen in fig. 5.1c, the difference in performance of both systems stays almost the same giving indications of clear gains in changing the PCIe from 2x1 to 3x16. M19 indicates that the difference drops at higher values in this graph, but extended tests show that this does not continue and stays above 500 elements per second difference. The conclusion is that bigger models such as M17 have lower throughput and have less difference between the PCIe connections but the performance difference stays the same.

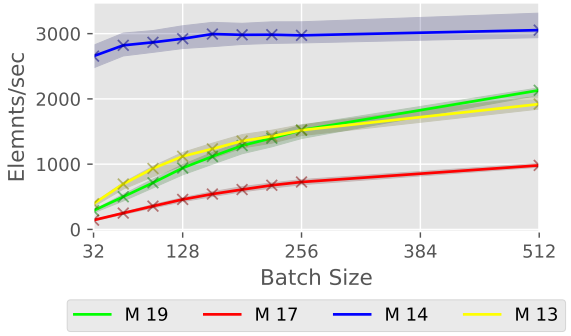
Another experiment that changes the parameter from *network size* to *batch size* has been conducted. The results are seen in fig. 5.2. The figures show that increasing batch size improves throughput. This is because GPUs are able to highly utilize their many CUDA and tensor cores increasing the degree of concurrent work done. The improvement is especially obvious in the LSTM models, where LSTM layers are dependent on previous time steps' output resulting in poor parallelization unless batched. The difference between the throughput of M19 and M13 is demonstrating the effect of using convolutions. M13 only has a single LSTM with no convolution and has almost the same throughput as M19 with three LSTM layers and a single convolution layer. This point is further demonstrated by comparing M17 and M19 where the throughput is significantly worse for M17 no matter the size of the batch.

The throughput of all models increases as the batch size increases although the throughput of the LSTM-based models are improved relatively more compared to models without LSTMs.

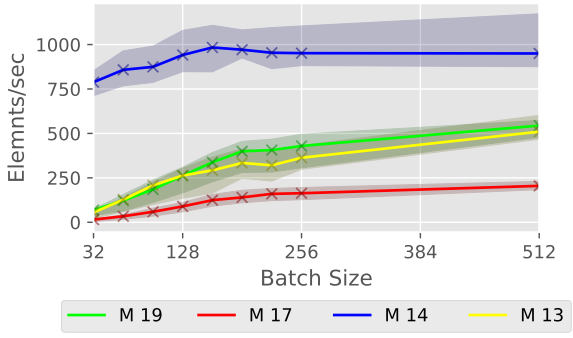
Based on the results here, it is concluded that small network sizes with high batch sizes give the highest throughput and that high stride values improve throughput. The limit of the batch size is given by the



(a) PCIe2x1 Performance.

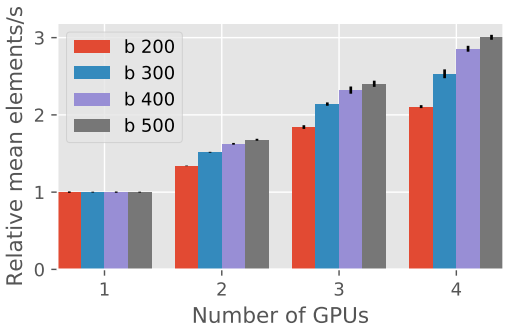


(b) PCIe3x16 Performance.

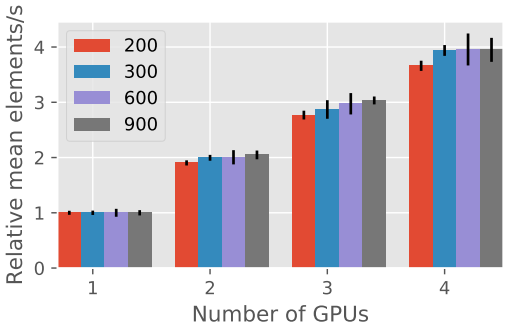


(c) Performance Difference.

Figure 5.2: Batch size training throughput experiment.



(a) RebelRig Scaling GPU.



(b) Sim Scaling GPU.

Figure 5.3: Multi-GPU throughput.

memory available on the graphics cards, that in the case of RTX 2070 is 8GB (7952MB).

2 Multiple GPUs

Using the model described in chapter 4 we scale using data parallelism essentially copying the model to each of the GPUs used having a data pipeline for each. This experiment is conducted on 50 batch iterations using a character based M19 with a network size multiplier of 800 with 10000 training elements.

Figure 5.3 and table 5.1 show the performance increase using multiple GPUs with different batch sizes on both RebelRig in fig. 5.3a and Sim in fig. 5.3b. The batch sizes of 500 on RebelRig and 900 on Sim maximises memory usage on the GPUs and crash the program if set higher.

Results show that using more GPUs and/or larger batch sizes increase throughput during training. The increase in performance is especially clear on RebelRig

	GPUs	batch	elm/s	relative
Rebel	1	200	395.5	1.0
	4	200	833.2	2.1
	1	500	482.0	1.0
	4	500	1448.2	3.0
Sim	1	200	318.2	1.0
	4	200	1164.8	3.7
	1	500	388.6	1.0
	4	500	1530.1	3.9
	1	900	390.7	1.0
	4	900	1542.65	3.9

Table 5.1: Multi-GPU throughput values.

fig. 5.3a. It is also the case for Sim fig. 5.3b where there is an increase in performance increasing batch size above 200 but no increase above 300.

In RebelRig's case, the performance is increased by a relative factor of 3 with a batch size of 500 when the number of GPUs are changed from 1 to 4. Sim increases to 3.9 times performance when using 4 GPUs. Figure 5.3b sometimes show higher performance than the number of GPUs, where the plotted relative performance is higher than the number of GPUs used. This is because of uncertainties in the measurement since the values are only based on 50 batch iterations. All the average values calculated were close to but below the relative number of GPUs.

In table 5.1 a single RTX 2070 performs better than the GTX 1080Ti. Two possible reasons could be: First, the RebelRig only uses a single CPU, and does not suffer from dual socket transportation of memory. Second, the clock speed of the CPU on RebelRig is much higher at 4 GHz compared to the 2.6 GHz on Sim.

The findings are similar to other papers where Deep Speech 1 [Han+14, p. 5] describes that they find smaller batches become memory-bandwidth limited. Krizhevsky [Kri14] use up to 8 GPUs (K20) and show speedups of 1.9 for two GPUs 3.7 for 4 and 6.3 for 8 GPUs are presented using PCIe 2.0 x16 to PCIe switches each connected to 4 GPUs. Strom [Str15] distributed the computation across compute nodes on Amazon Web Service (AWS) using GPU instances with a single GPU. The relative speedup in that paper using 5 nodes was 4.3 and with 80 nodes 54 times speedup

The throughput is only as fast as the slowest GPU because the batched result is calculated based on the result of all individual GPUs [Dea+12]. This means

nothing for the Sim setup since all GPUs are exactly the same, but it has a major impact on the performance of RebelRig where the one GPU connected with PCIe3x16 is slowed down by the others.

6 Served Model Experiments

The performance of the served model is evaluated based on latency, throughput and quality of output. The model is served on the *Sim* machine without hardware acceleration and therefore only using CPU. This was chosen because systems such as SMLTA [Gro19] performs well when served on CPU, but scaling the number of concurrent users would benefit from batching and offloading the calculations to GPUs. The experiments and measures used are described in the following sections.

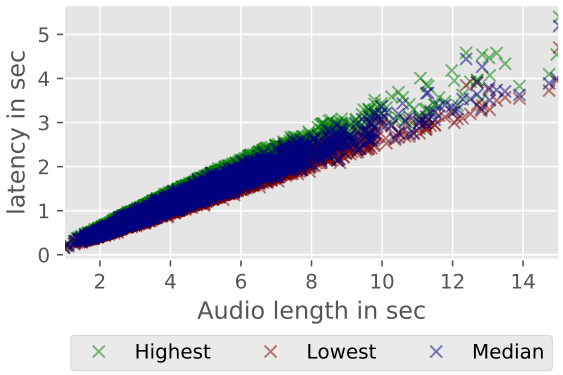


Figure 6.1: Latency scaling audio length M33.

1 Latency

Latency is the time it takes for the client to be serviced by the model [Gre13, p. 16]. This means that we need to measure the time from a request is sent by the client until a response is received by the client. We are interested in finding how the latency changes with increasing number of concurrent requests. Therefore, the entire dataset is sent sequentially as requests first with a single requester, then with double the amount of concurrent requesters and so on doubling until a bottleneck is found. The experiment is repeated to make sure that the results are stable. The concurrent requesters are simulated using a python multiprocessing pool, with the number of concurrent users as the number of threads allocated for the pool.

For each request, the following is written to a CSV-file: the latency in seconds, the target sentence, the result sentence, the filepath, the loaded audio size in bytes and the file size in bytes. By writing the file size and loaded audio size to the result file, it is possible to see how the latency is related to the size of the input to the model and the duration of the audio clip. These results are also used when measuring the quality of the results.

1.1 Latency Results

Figure 6.1 and fig. 6.2 are plots of the highest, lowest and median latency of different audio file lengths with sequential requests. The plots show a linear relation between the duration of processing and the length of the sound files. For all models in this project, the latency is lower than the length of the audio and the linear relation is maintained. As the audio length increases, the relative latency per second of audio decreases. This is likely due to overhead in sending and receiving requests. Additionally, it is observed by comparing fig. 6.2 to fig. 6.1 that simpler models with fewer and smaller layers, such as M20 with three LSTM layers, have lower latency and complex models, such as M33 with five LSTM layers, have higher latency.

Figure 6.3a shows the relation between the number of concurrent users and the latency in seconds for M33. The plot shows the upper and lower quartiles in the shaded areas and the median latency on the line. The latency is stable when the number of concurrent requests is less than the number of logical cores of 24. When the number of concurrent requests increases from 2^4 to 2^5 thereby exceeding the number of logical

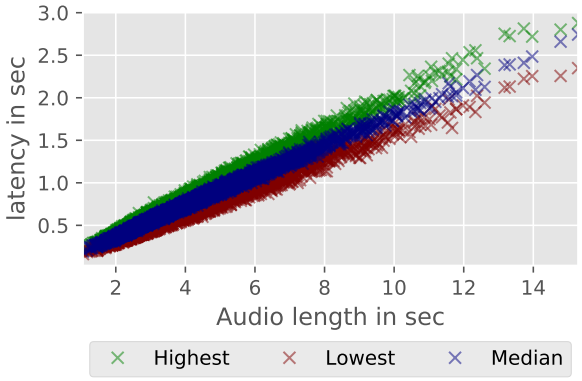


Figure 6.2: Latency scaling audio length M20.

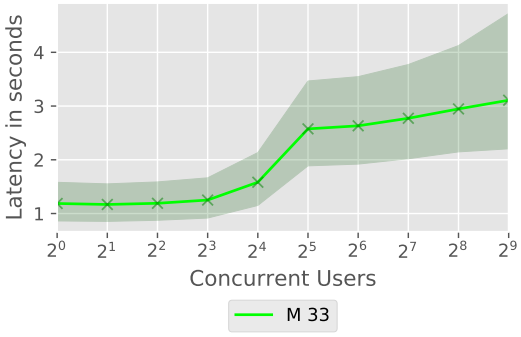
cores, the latency quickly increases to almost 3 seconds median latency. Subsequently, the median latency steadily increases between $2^5 = 32$ and $2^9 = 512$ concurrent users. The number of concurrent users was scaled above 2^9 but suffered severely when exceeding 1024 because the RAM was filled. A reason why the RAM was filled could be the work queue was being filled faster than requests were handled.

Because this experiment uses the *Sim* machine as both requester and server without pinning the server and requester Docker containers to specific cores, the system has more context switches and higher memory usage when the number of requests grows compared to a setup where the requester and server are either pinned to specific cores or are on separate machines. This means that the actual performance without handling the requests concurrently on the same machine could be better than the one shown in fig. 6.3a.

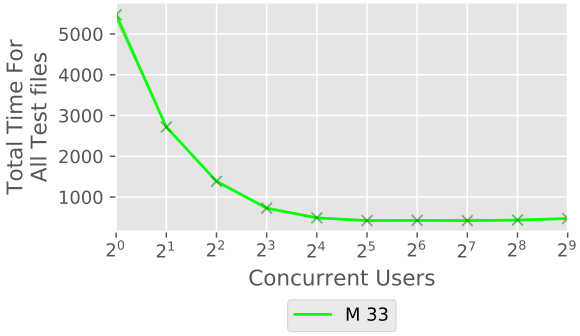
2 Throughput

Throughput is the volume processed over time [Gre13, p. 27], hence it gives us the amount of audio we can process within a certain time limit. The experiment setup is similar to the latency experiment, but with the addition of measuring the time it takes from the first audio request until the last response is received. As in the latency experiments, the number of concurrent requests, the experiment iteration, and the elapsed time are written to a CSV-file. The maximum of concurrent users was again set to 512 because if increased the RAM would be filled.

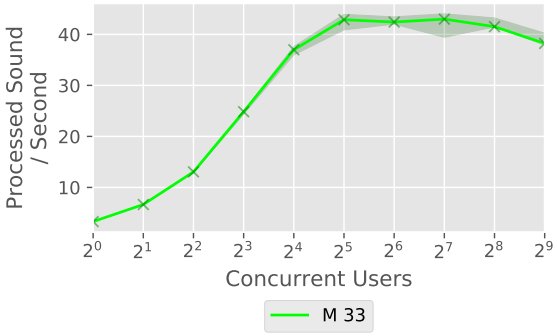
To calculate the throughput, the total time it takes to



(a) Latency scaling concurrent users.



(b) Total time for all test files.



(c) Throughput.

Figure 6.3: Concurrent users latency and throughput.

process all the files is first measured. This is a measurement that includes the entire setup of concurrent users in the thread pools and the timer is stopped when all requests have been processed. Figure 6.3b shows the total time to process all the files in the test dataset with increasing amount of concurrent users. Taking a closer look at the results, it took 5400 seconds to process the 18100 seconds of sound with one user. Increasing the number of concurrent users above 32 maintains the performance but it does not improve it.

The throughput is then calculated by:

$$\text{Throughput} = \frac{\text{Total Time of Files}}{\text{Total Processing Time}} \tag{6.1}$$

This relation is shown in fig. 6.3c. The graph shows that it is possible for the system to process 40 seconds of sound per second. This means that the system can handle 40 real time producers of sound concurrently without building up a backlog.

All the latency and throughput experiments fits well with the fact that the system has 24 logical cores. Increasing the number of concurrent requests above this point does improve performance and the results show that the system is able to handle more with no significant loss to performance.

3 Quality

The two previous sections describe how fast the ASR models return results and how well the system handles requests. This section describes the quality of the models, which relates to how close the output of the ASR system is to what was actually said in the audio given.

3.1 Metrics

The quality of the model is evaluated by comparing the output sentences to the target sentences. Several metrics evaluating the quality quickly and systematically exist. We decided to use WER, CER and BLEU and they are described in further detail in the following subsections.

The quality experiments are conducted based on the output and target sentences from one of the latency experiments. The output only needs to be retrieved once since the model is deterministic, hence the output will always be identical given the same input with the assumption that no exceptions occur. For each target and result sentence pair, the following is written to a CSV-file for later analysis: filepath, WER, CER, BLEU, target sentence and result sentence.

CER

A common metric in ASR is *Character Error Rate* (CER), which is used in Deep Speech 2 [Amo+15], Cold Fusion [Sri+17] and in "End-to-end attention-based large vocabulary speech recognition" [Bah+16]. These papers do not specify how they calculate CER, but define it as similar to WER just with characters instead of words. The WER is based on the number of insertions, deletions and substitutions, which on a character-level is called an *edit distance* or *Levenshtein distance* [Nav01].

The Levenshtein distance is calculated by counting the number of insertions, deletions and substitutions relative to the maximum number of edits denoted by $\max(|x|, |y|)$, hence the number of edits $e(x, y)$ of a sentence x and a sentence y has the property $0 \leq e(x, y) \leq \max(|x|, |y|)$ which means that the Levenshtein distance is:

$$CER = d(x, y) = \frac{e(x, y)}{\max(|x|, |y|)} \tag{6.2}$$

The Levenshtein distance $d(x, y)$ consequently has a value between 0 and 1, where 1 means that the sentences are far from each other and 0 means they are identical.

We used the Python implementation of Levenshtein distance from the *text distance* package by Orsinium [Ors19].

WER

The most common quality evaluation metric in ASR is *Word Error Rate* (WER) [YD15; Amo+15]. WER is based on the number of insertions, deletions and substitutions necessary to convert the result sentence R to the target sentence T. WER can be described as [Kin18; VD12]:

$$WER = \frac{I + D + S}{W} \tag{6.3}$$

In this function, I is the number of insertions, D is the number of deletions, S is the number of substitutions and W refers to the number of words in the target sentence T. Since the division is the number of words in the target sentence the score can become above 1 if the output sentence is longer than the target. A more precise definition of WER is based on the word-level Levenshtein distance, hence WER can be defined as [PN07]:

$$\text{WER} = \frac{1}{|T|} \sum_{k=1}^{K} d_L\left(\mathrm{T}_k, R_k\right) \tag{6.4}$$

Where d_L denotes the word-level Levenshtein distance, $|T|$ is the length of the target sentence T and K is the number of sentences in T and R which in our case is always 1, hence eq. (6.4) could be rewritten as:

$$\text{WER} = \frac{d_L\left(\mathrm{T}_k, R_k\right)}{|T|} \tag{6.5}$$

We used the implementation of WER from the Python package *Jiwer* by Vaessen [Vae19].

BLEU

Within the field of *Machine Translation* (MT), *Bi-Lingual Evaluation Understudy* (BLEU) is a widely used metric [HDA11; Pap+02; CC14]. The purpose of BLEU is to evaluate machine translations quickly, inexpensively and independent of language [Pap+02]. It does so by taking the geometric mean of n-gram precisions between the target sentence and the produced sentences [HDA11; CC14]. Geometric mean is the n-th root of the product of n numbers, opposed to arithmetic mean that sums and divides with the length. BLEU correlates well with human judgments on the document level, but on the sentence level it has poor correlation. This problem is alleviated by introducing smoothing functions and brevity penalty [CC14].

Although BLEU is mainly used in MT, it is also widely used within the ASR-related field of *Speech Translation* (ST), where speech in one language is recognized and translated to text in a different language [HDA11]. In ST problems, WER sometimes leads to worse translations even if it is only used to evaluate the ASR module of the ST system [HDA11]. It is not common to apply BLEU to systems that only generate sentences from speech, but because of the promising results within ST and because it is an n-gram based precision metric, we decided to apply it in our project.

A BLEU score ranges from 0 to 1 where lower scores signify low quality and higher scores signify high quality [Pap+02]. However, with certain smoothing techniques, for instance smoothing technique 5, the score limit exceeds 1. The precision part of the BLEU score P is calculated with a result translation sentence T, which is equivalent to our actual output sentence; a reference sentence R, which is the ideal output of our system also called *target*; and the maximum number of grams N, in our case 4:

$$P(N, T, R) = \left(\prod_{n=1}^{N} p_n\right)^{\frac{1}{N}} \tag{6.6}$$

where:

$$p_n = \frac{m_n}{l_n} \tag{6.7}$$

with m_n as the number of matched n-grams between translation sentence T and the reference sentence R and with l_n as the total number of n-grams in the translation sentence T [CC14].

The poor correlation between human judgments and BLEU on the sentence level occurs when an n-gram precision of a sentence is 0, because according to eq. (6.6) and eq. (6.7) the BLEU precision score $P(N, T, R)$ of the entire sentence evaluates to 0 when a single m_n is 0. This is alleviated by introducing smoothing techniques. Chen and Cherry [CC14] compared seven different smoothing techniques including three new techniques specifically developed to correlate better with human judgments. Technique number seven correlated well with human judgment in the experiments conducted by Chen and Cherry, hence this technique was chosen for this project.

Technique seven is a combination of two other smoothing techniques, which is technique 4 and technique 5 in "A Systematic Comparison of Smoothing Techniques for Sentence-Level BLEU" [CC14].

In smoothing technique 4, m_n is replaced by a geometric sequence covering the cases where $m_n = 0$. The algorithm for the calculation is in algorithm 7.

Algorithm 7 Smoothing technique 4 from [CC14]

```
invcnt ← 1
for n in 1 to N do
    if m_n = 0 then
        invcnt ← invcnt × (K / ln(len(T)))
        m'_n ← 1 / invcnt
    end if
end for
```

In this algorithm, m'_n replaces m_n, K is a constant set empirically, T is the translation sentence and $\text{len}(T)$ is the length of sentence T. The technique assigns a smaller smoothed count to shorter sentences.

Smoothing technique 5 takes the averages of m_{n-1}, m_n and m_{n+1}, hence the modified n-gram match count m'_n is:

$$m'_n = \frac{m'_{n-1} + m_n + m_{n+1}}{3} \tag{6.8}$$

starting with $m'_0 = m_1 + 1$. The idea of this smoothing technique is that m'_n for all similar values of n should be similar. For instance, if the bigram count is high then the trigram count should also be high even though no actual trigrams exist. This is based on the idea that a bigram is an incomplete trigram, hence the bigram count should add to the trigram count.

Smoothing technique 7 combines the two other techniques by first calculating m'_n with technique 4 and then taking the average of all modified and unmodified n-gram match counts as in technique 5. In this way, the smoothing technique has the best of both worlds with smaller smoothed counts for shorter sentences and similar smoothed counts for similar n-grams.

The brevity penalty $BP(T, R)$ of the result translation sentence T and reference/target sentence R is:

$$\text{BP}(T, R) = \min\left(1.0, \exp\left(1 - \frac{\text{len}(R)}{\text{len}(T)}\right)\right) \tag{6.9}$$

where $\text{len}(R)$ is the length of sentence R and $\text{len}(T)$ is the length of sentence T [CC14]. If the length of the result translation sentence T is shorter than the target sentence R then $\text{BP}(T, R) < 1$ and if T is longer than or equal in length to R then $\text{BP}(T, R) = 1$.

The final BLEU score is then calculated by multiplying the precision P with brevity penalty BP [CC14]:

$$\text{BLEU}(N, T, R) = P(N, T, R) \times \text{BP}(T, R) \tag{6.10}$$

The BLEU score will always be greater than 0 and result sentences shorter than the target sentence will get lower BLEU score than result sentences longer than or equal in length to the target sentence. The BLEU score is calculated with the NLTK Translate package, where the seven smoothing techniques and brevity penalty are implemented [Pro18]. The maximum number of n-grams is set to 4, which is the default in the NLTK Translate package.

3.2 Results

This section defines the results of the quality experiments of the models. The objective is to achieve a performance that is close to the related systems. The results achieved in related work are seen in table 6.1. Two WERs are defined for each system where possible. The first WER is from a clean dataset and the second WER is from a noisy dataset where data augmentation has been applied. The experiments in this project are based on Mozilla Common Voice, which is smaller than the datasets used in related work. The datasets used in some of the related work contain over 2000 hours of speech, but some papers also publish performance on smaller amounts of data which is included as well. The speech clips are of varying lengths ranging from single utterances to continuous speech.

It is expected that the WERs of this project would be higher if the models were applied to the larger datasets used in the related work since the vocabularies in these datasets are larger.

The sizes of the models in the related work also exceed what would fit the hardware setup for this project with our batch sizes, hence the sizes of the model in this project are smaller.

The following subsections look at the effect on quality when the AM is changed based on the following: character-based or word-based output; LSTM or BiLSTM layers; Mel, LogMel or MFCC features.

Acoustic Model Output & Direction

The results of four different models are seen in table 6.2. In this table, the C is an abbreviation of characters and W is an abbreviation of words, meaning that the model names and BLEU scores with C are mapping characters and W are mapping words. The model settings for the character-based models are a single convolution with width 8, stride 2 and three LSTMs of size 1024. The convolution of the word-based models are width 16, stride 8 and each of the three LSTMs has size 1024. All models are trained using batch normalization and dropout of 0.5 with a dataset cutoff length of 300 MFCC.

First comparing the performance of the word-based models to the character-based models, it is seen that the word-based models outperform the character-based models in all metrics. The difference between the models are smaller when looking purely at the CER. The reason for this is that the character-based models are having trouble spelling the words correctly all the way through the words. A single wrong character in a word will significantly increase the WER, but only affect the CER slightly. This means that the character-based models may be penalized too much by the WER metric

Model	Hours	WER (Clean)	WER (Noisy)
Deep Speech 1 [Han+14, pp. 7-8]	300	25.9	n/a
Deep Speech 1 [Han+14, pp. 7-8]	2300	16.0	19.1
Deep Speech 2 [Amo+15, p. 17]	120	29.23	50.97
Deep Speech 2 [Amo+15, p. 17]	12000	8.46	13.59
Listen, Attend, Spell [Cha+15, p. 7]	2000	10.3	12.0
Attention by Enforcing Monotonic Alignment [Raf+17, p. 7]	80	16.0	n/a
CLDNN-HMM [Cha+15, p. 7]	2000	8.0	8.9
PAPB for Seq2Seq ASR without LM [Bas+19, p. 5649]	80	10.8	n/a
PAPB for Seq2Seq ASR with LM [Bas+19, p. 5649]	80	3.8	n/a
A spelling correction model for End-to-end ASR (Upgraded LAS) [GSW19]	960	4.28	n/a

Table 6.1: Best quality results of related work.

Model	Type	Size	Batch-size	WER	CER	BLEU_C	BLEU_W
M 30	LSTM_C	1024	256	99.9	53.7	26.4	9.3
M 29	BiLSTM_C	1024	256	78.0	32.5	50.9	20.7
M 30	LSTM_W	1024	800	36.9	29.6	72.6	49.8
M 29	BiLSTM_W	1024	800	37.3	30.9	68.9	50.7

Table 6.2: Quality results of character-based and word-based models with LSTMs and BiLSTMs.

compared to how much a single wrong character will affect a human's understanding of the text. The BLEU score is meant to be closer to humans' perspective of what constitutes a correct and comprehensible text. Looking at the character-based BLEU-score it is seen that the BiLSTM_C is close to the word-based models, hence it is close to the labeled text regarding the meaning, although the word-based models still have better scores.

The conclusion of comparing the word-based models to the character-based models is that the character-based models are not able to spell most words correctly and that it is more difficult for the character-based models to learn because of the increased complexity of outputting long sequences of letters instead of outputting shorter sequences of words.

The latter conclusion is further backed up by the training duration, which was significantly higher for the character-based models than the word-based models. Furthermore, the loss value during training was higher for the character-based models meaning that it underfits the training set. The underfitting could have been impeded by producing more frames in the feature extraction by reducing the frame step size. Another option is to increase the amount of training data either by more recordings or adding data augmentation.

Regarding the differences between using LSTMs and BiLSTMs, the measurements tend to be better when using BiLSTMs although the difference is not as pronounced as the difference between character-based and word-based models. The BiLSTM_C model has a significantly better WER and CER, but the BiLSTM_W model actually has a worse WER, CER and BLEU_C compared to LSTM_W.

The conclusion of this is that the quality improvements of using BiLSTMs instead of LSTMs is small or non-existent in this setup. Based on these initial results we decided to further investigate and improve the LSTM_W model for the next experiments.

Feature Extraction

Based on the initial results and many experiments where the batch size, feature extraction, model size, LSTM parameters, and Beam Search were changed we ended up with a model equal to LSTM_W from initial results performing radically better.

The major changes that improved performance was done in the AM. First, a batch normalization layer before the convolution was added. Second, peephole

Model	Layers	Feature	Size	WER	CER	$BLEU_C$	$BLEU_W$
M 31	3	Mel	1600	14.8	10.9	96.4	84.6
M 31	3	Log Mel	1600	13.7	10.3	97.2	85.9
M 31	3	MFCC	1600	14.4	10.8	96.6	85.2

Table 6.3: Quality results of FE experiments with word-based uni-directional LSTM models.

connections were added to the LSTMs. Third, the LSTM size was increased to 1600 to avoid the under-fitting problem from previous experiments. Fourth, the stride of the convolution was reduced to two which enables the model to output more words. Finally, a forget-bias equal to the dropout rate was set on the internal state of the LSTMs. The training procedure was also changed so that when the validation loss is not improved during the last 1000 iterations, the checkpoint of the model is loaded and the training continues with a lower dropout and forget-bias. The dropout and forget bias was set to 50% at the start of training and was decreased two times first to 30% at around the 2500 iteration and then to 0 at the 9400 iteration. When serving the model the Beam Search width was reduced to 1, effectively changing the Beam Search algorithm to a best-first search. This reduction reduces the latency of the served model but with a chance of producing lower quality results.

The quality results of the model with various feature extraction methods are seen in table 6.3. The WER, CER and BLEU for the three variations of M31 are very similar, but in all metrics the Log Mel model is slightly better than the two other models. The reason that the Log Mel model performs better than the MFCC model could be because some of the important information from the sound are discarded when using MFCCs. As mentioned in chapter 2 section 3.1, MFCCs consists of 13 features per frame whereas Log Mel consists of 64 features per frame, hence Log Mel could potentially carry more information than MFCC. It is important to note that more features are not necessarily better, which is evident from the quality difference between Mel and Log Mel which both have 64 features per frame. Using Log Mel instead of Mel lowers the WER from 14.8 to 13.7, hence doing FE does not only lower the amount of data but also has an impact on the final quality of the output.

Number of Layers

The final modification of our models is testing with a different number of LSTMs layers. Table 6.4 shows that when the number of LSTM layers in the model is increased to seven, while decreasing the layer size so that the total number of weights in the model is approximately the same, the results become even better.

Comparing the WERs of the models presented in this section to the WERs in table 6.1, it can be concluded that the quality of the best models in this project performs similarly to related work that do not use a language model. The datasets used in the related work are different and several models use further processing after the AM such as language models, hence the performance scores cannot be compared directly.

If we compare the models in table 6.4 to the models that use the same amount of training data as our models in table 6.1, then it is seen that the models in our project performs better than the related work with the same amount of training data. This could be because the models trained in the other papers have a too high capacity for the size of the reduced dataset. For instance Deep Speech 2 [Amo+15] builds a model with a capacity suitable for a large training dataset, hence when the training dataset is reduced to 1% of this dataset, the model overfits which makes the WER worse than it would have been if they built the model specifically for the small training set.

Human Evaluation of Challenging Input

Looking further into the results of M33 with Log Mel FE, 40 random files that have 0 WER and 40 files that have 1 WER were selected for a manual investigation of challenging input. These files were all listened to and classified with the different labels shown in table 6.5.

The first labels are *Male*, *Female* and *Unk*. Mozilla have noted that 41% are male and 10% are female in their dataset [Moz18], the remaining 49% are unlabeled. This matches the distribution found in both the correct and the wrong classifications in the results. The results show that the system does not appear to be biased with regards to gender since it performs similarly on both genders.

The *Noisy* label is the count of files with a large degree of noise in the recording. Here it is evident

Model	Layers	Feature	Size	WER	CER	$BLEU_C$	$BLEU_W$
M 32	1	Log Mel	4800	17.4	14.1	92.2	81.1
M 31	3	Log Mel	1600	13.7	10.3	97.2	85.9
M 33	5	Log Mel	960	10.5	7.8	100.9	92.3
M 34	7	Log Mel	685	14.1	10.4	97.2	86.4

Table 6.4: Quality results of different number of uni-directional LSTM using Log Mel.

that it is harder to classify the recording with noise. It is expected that noisy input impedes the quality of the model, which was also reported by Deep Speech 2 [Amo+15] and LAS [Cha+15] as seen in table 6.1. By including more noisy recordings in the dataset or by adding noise to the existing recordings, the model's quality performance on noisy input could be improved. The results presented here indicates that the WER could be reduced by doing so.

The *Accent* label describes whether the speaker in the recording has a heavy accent. A heavy accent is defined as a pronunciation that significantly deviates from the norm of the dataset. The count shows that it is difficult for the system to convert speech with heavy accents to text. The training dataset contains several different accents, hence the model is trained to work on different accents, but the results here show that the model has not been able to adapt completely to all accents. This problem could be alleviated in two ways. One way is by sampling the training dataset based on accents, so that an equal amount of all different accent categories are given as input. In this way the model would not be trained to recognize one accent better than another accent. The problem of this approach is that if the model has to generalize to all accents, the average WER may be lower. Another way to alleviate the problem could be by building separate models for different accents. In this way, a user of the system could choose the accent before sending data through the system. The problem with this approach is that the accent would have to be labelled before converting speech to text, hence it is not speaker independent as defined in chapter 2 section 2.

Next label *Fast* describes if the sentence is uttered fast. It is evident that this is a challenge for our system. The challenge could be handled by decreasing the frame step size during FE, so that more frames are generated from the same files. In this way, the acoustic model could have a more fine-grained processing of the input words even with fast speakers. Another idea to solve the challenge is to use data augmentation of the training dataset so that the speech pace is accelerated, enabling the model to learn the faster utterances with more data.

	Correct	Wrong
Male	29	27
Female	11	10
Unk	0	3
Noisy	2	7
Accent	0	6
Fast	4	14
Correct	40	37

Table 6.5: Features from 40 random files with correct output sentences and 40 files with the completely wrong output sentences.

Finally, *Correct* describes if the sentence spoken is equal to the labeled sentence. This is wrong in three files of which two do not contain sound and one contains someone yelling *"hosar uda"* whereas it should have been *"a line of flame high can be seen in the atmosphere"*.

4 Efficiency versus Quality

An efficient model producing high quality results is preferred over an inefficient model producing low quality results. Often the most efficient model is not the one producing results of highest quality. Phrased differently, there is a trade-off between efficiency and quality. By using the metrics defined in the previous sections, this trade-off can be quantified.

The latency and quality of selected high performance models are visualised in fig. 6.4. The figure is based on data from the experiment where 24 concurrent requests are sent to the model. All elements of the test dataset are sent to the model and from this the average latency in seconds and WER are calculated. The models have the same sizes as defined in table 6.4, which means that the five-layer models have lower size per layer than the

single-layer model.

The figure shows that M33 with Log Mel FE is outperforming the other models on quality with a lower WER but M34 is slightly more efficient with a lower latency. This means that M33 is the preferred model if a low error rate is needed and M34 is preferred if low latency is needed.

The three variations of M31 are positioned in a cluster in fig. 6.4. Looking closer at the models, it appears that the WER is distributed as defined in section 3.2 and the latency is different depending on the FE. Using Mel, which is the simplest type of FE, the latency is highest. Using Log Mel, which has the same amount of extracted features as Mel, the latency is slightly lower. Using MFCC, which is the FE with fewest extracted features, the latency is lowest. This suggests that the FE method influences the latency of the model, where more extensive FE with fewer extracted features lowers the latency. It should be noted that the latency differences are very small and so are the WER differences, hence FE does not have the most significant impact on neither quality nor latency.

M32 with Log Mel FE has only a single LSTM layer, but it has both the lowest latency and highest WER of the models in fig. 6.4. It could be expected that models with a lower number of layers have lower latency, but in this case it seems to be the opposite. The reason for this could be that the size of the LSTM has been increased disproportionately compared to the models with more LSTM layers, but looking at the size of the file containing the saved model parameters it is concluded that the single-layer model has more saved parameters than the other models. The WER of M32 is lower than the models with more layers, hence the extra parameters of M32 do not result in better quality. The conclusion of this is that more parameters does not necessarily give better quality, but it increases the latency.

The M34 experiment was conducted after looking at the tendency from models M31-33, indicating that more smaller layers had both better latency and quality. It is shown in fig. 6.4 that this tendency does not continue. The latency falls at the cost of higher WER when increasing the number of LSTM layers from five in M33 to seven in M34.

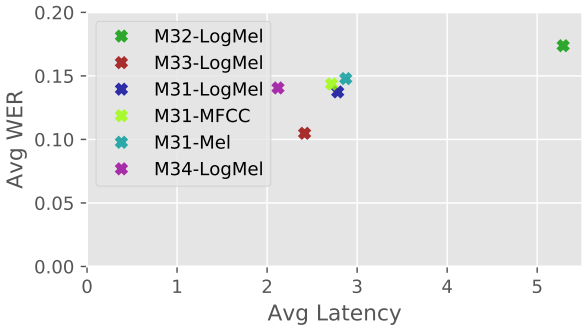


Figure 6.4: Latency vs WER of selected models.

7 Future Work

This thesis has described the design and implementation of an ASR system. Several ideas on how to improve the implementation or experiments of the system emerged throughout the project. This chapter describes the best ideas for future work within ASR that could improve the performance or evaluation of the performance.

The chapter is divided into different sections that each constitute a category of future work. The first section is *Quality Improvements*, which includes ideas that could further improve the performance of the model by achieving a lower WER and by being able to handle more diverse and noisy inputs. The second section is *Training Efficiency* where ideas for improving efficiency of training the models are presented. The third section is *Serving* which has suggestions of how to achieve lower latency for requests to the served model and how to improve the scalability with regards to the number of concurrent users. The final section is *Measurements* describing measurements and visualizations that could improve the understanding of performance of the models.

1 Quality

The first thing one could do to improve the quality of the current best performing word-based model would be to continue trying different *parameter* settings. Doing so is time consuming as it takes up to or longer than a day to train a single model, hence the parameters chosen for the model and training process should be well thought-through. A possible solution is to train the model on smaller amounts of data thereby reducing the time it takes to fit the model to the training data. The downside to this is that models with large capacities tend to overfit small amounts of training data. The solution to the overfitting is smaller models with smaller capacity, but then the models would not fit to the full dataset making the smaller models measurements inaccurate for the larger dataset. The tendency to overfit with a model that have a large capacity can for instance be seen in Deep Speech 2 [Amo+15], also seen in table 6.1, where the large model performs worse when reducing the amount of training data without changing the model.

A second element to work with would be *data augmentation* as mentioned in chapter 2 section 4.7. Most of the related work, for instance Chan et al. [Cha+15], rely on data augmentation for improvements in performance and increasing the robustness of the system.

A third option would be to improve the output hypothesis with a language model, mentioned in chapter 2 section 3.3. Most related work [Amo+15; Han+14; Cha+15] show performance measurements with and without language models applied. The language model clearly improves the quality of their WER. This indicates that our model would benefit from a language model as well. For the language model to work with our setup, it could be included in the Beam Search replacing the default heuristic function. A concrete modification of Beam Search and how to use it is described in Scheidl, Fiel, and Sablatnig [SFS18] with an open-source implementation on GitHub [Sch19]. The solution describes a heuristic function that is based on a combination of the output of the inference model and the language model score. With this solution, experiments determining the efficiency of different language models and the resulting WER could be conducted.

A fourth option would be to lower the difficulty of the training. Deep Speech 2 [Amo+15] describes the difficulty of a file as the length of the file, categorising shorter audio files as easy and longer audio files as harder. The model initialization improves if trained first on easier elements and then afterwards train on progressively harder elements. Deep Speech 2 names this learning strategy SortaGrad, and an implementation of this technique is to: First, sort the first few

iterations of the dataset based on the difficulty, which is the length of the audio file. Second, batch similar audio length samples together while training. Doing this will result in faster training because there is less padding on similar length files. Furthermore, it improves the convergence of the model's parameters because it trains on the easy elements first [Amo+15, p. 7].

Further potential improvements of the quality could be possible if the model was changed to use other approaches. One of these approaches could be the attention based model mentioned in chapter 3 [Bat+17] and SMLTA [Gro19]. The only changes needed for this would be modifications of the inference model to an attention based model and replacing the loss function. Other newer ASR systems just published in this field at ICASSP [ICA] show a trend towards these types of networks as well as adversarial networks for generating challenging inputs for the ASR systems.

2 Training Efficiency

Computations on GPUs have to be optimized differently than optimized operations on CPU. This resulted in a performance drop in this project where we used CPU optimized LSTMs trained on GPUs. While it is briefly mentioned in chapter 4 section 1.2 that we did try cuDNN LSTMs for better optimized GPU performance we did not use it in the end.

Another training efficiency improvement is to reduce the amount of padding used on training elements when batching elements together. The strategy used in this report was to pad all elements to the same length. All elements were padded to 300 frames length and each frame was 32ms resulting in 9.6 seconds for each file. Comparing this to the average length of the sound files of 4.1 seconds it can be concluded that 5.5 seconds of padded values were processed too much per file on average. One strategy could be to batch and pad similar lengths together to smaller sizes reducing the amount of padding like in the SortaGrad technique from Deep Speech 2 [Amo+15].

Mixed position training mentioned in chapter 4 section 3.4 is also a potential improvement for the training efficiency [Mic+17]. The technique was employed in one of the inference models, but with low performance most likely due to incorrect implementation. The reason why this was not further investigated is because mixed position training does not improve performance on the GTX 1080 Ti cards [Per19], but for future work it makes sense since the GPUs are moving towards greater support for the 16-bit float operations [NVI18].

3 Serving

The serving of our model is using the basic features of TensorFlow Serving, and could be further improved by applying some of the more advanced features such as hardware acceleration, batch dispatching, and service distribution.

Hasan [Has19] describes how to improve TensorFlow Serving by compiling it. Compiling does not in itself improve performance, but depending on the CPU and compile flags the performance could be improved. All of the flags mentioned here are CPU instruction set extensions that allow for Single Instruction Multiple Data (SIMD) operations. One of the compile flags is Advanced Vector Extensions (AVX) or AVX2 that allows optimized vector calculus which is used a lot in neural networks for multiplying matrices. The TensorFlow installed from Docker or Pip for Python, which is used in this project, was compiled with the AVX flag. Another interesting compile flag is Fused Multiply-Add (FMA) which executes a multiplication and an addition together. Another optimisation is Streaming SIMD Extensions (SSE) flags, specifically SSE 4.1 and SSE 4.2 since these are supported by RebelRig. Since the CPU on Sim uses the Ivy Bridge micro architecture, it can only use the AVX optimisation while RebelRig is able to potentially benefit from all of the above-mentioned compile flags with its Skylake micro architecture.

Another improvement of TensorFlow Serving is to include batching of elements from requests following the guidelines from TensorFlow [Ten18] and Hasan [Has19]. This exploration of batching on the served model could be done both on CPUs and GPUs and the experiment results could be compared to the serving on the CPU described in this thesis. Another way to further this chain of thought would be to distribute the serving for instance by load balancing between different machines serving the model. All of the above points would add to the scalability of the serving by improving throughput on the setup enabling more concurrent request.

Another direction would be to prioritise the latency of the served model. This could be done by optimizing the model for faster execution. One approach could be to reduce the size of the model layers thereby lowering the amount of calculations needed from one inference

maybe at the cost of lower quality.

It would also be interesting to look at streaming sound to the server, resulting in a stream of results in real time. The simple implementation of this would be to window the sound using an appropriate windowing strategy such as tumbling or session windowing, and sending each window to our model for inference [Ros18]. Tumbling windows are non-overlapping time or count based windows [Kip+17], that in this case would be based on a specific number of samples in the audio. This would take the input and slice it into equal size windows. This has the risk of cutting a window in the middle of a word. A more advanced windowing strategy that could avoid this problem while also increasing performance could be Session Windows. Session windows are windows that react on events by sending the window when specific events occur [Ros18]. This could be done by only sending sequences of frames that contained sound starting the window slightly before and ending it after. This could be done through a filter that would start and end windows if there is no amplitude in the recording. Taking this one step further the system could be trained for keyword spotting (KWS) instead to determine what to include in a request [CPH14]. This is the same technique employed by virtual assistants for instance Alexa [Ama], where "Alexa" is the keyword that the system constantly listens for.

Another direction would be to stop using the centralized server approach and look at deploying the model to mobile devices. This could be done using TensorFlow Lite [Ten19a] [ZK18, pp. 34-35]. This would remove the overhead of sending requests over the internet, but would suffer from the slower speed of the processing units on mobile.

4 Measurements

Using different and more measurements could provide insights into the models' performance and what steps could be taken to further understand the models.

To further the understanding of the effect of different parameters, the correct measurements have to be employed. A measurement that is not used is to plot and study the alignment of the outputs with the audio files. This is shown for instance in Battenberg et al. [Bat+17]. The way it is shown is the frames from the feature extraction are aligned with the the words or characters that are inferred in the model. This gives insights into when the models output the words compared to the sound in the input. This could provide useful information since the model's output is not necessarily aligned with the utterances.

Another direction for future work could be changing the model back to a character-based model. Related work almost exclusively make character-based models [Han+14; Amo+15; Cha+15; Bat+17; GSW19]. This is because a word-based model is limited to the specific vocabulary that it is trained to use. This means that word-based models receiving many out-of-vocabulary words is expected to have lower WER compared to character-based models.

The system is already able to switch between character-based and word-based output, but the current results from the character-based models are worse than the word-based ones. We believe that improvements in the output could be made with minor changes. The step size in the FE could be reduced, since most of the other papers use a step size of 10ms [Han+14; Amo+15; Cha+15; Bat+17; GSW19]. The frame size could also be changed to see the effect of doing so. This could be interesting, because the effect of different frame sizes in ASR models has not been reported in any related work.

8 Summary & Conclusion

This master's thesis describes the implementation of an automatic speech recognition (ASR) system. The thesis starts with a chapter providing the background of ASR. It describes the basic components of an ASR system, which is feature extraction (FE), acoustic model (AM) and language model (LM). An ASR system without FE and LM could be designed, where the AM is an end-to-end model taking audio as input and outputting characters or words. Different variations of this kind of model exist, for instance the LM could be omitted or the FE could be of different types. The difference between Mel, Log Mel, and MFCC is explained in the background chapter and this is later used in the design of the models. The AM has traditionally been based on statistical models, but research in recent years has focused on deep learning. The fundamentals of deep learning is explained, which includes Feedforward Neural Network (FFNN), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and Bidirectional LSTM (BiLSTM). Furthermore, different activation functions, Connectionist Temporal Classification (CTC) loss, different optimization algorithms, batching, regularisation, and the basic architecture of TensorFlow are explained.

After the background chapter, related work within the field of ASR is presented. Baidu has delivered some of the central contributions to recent research in ASR. The systems are referred to as Deep Speech 1 [Han+14], Deep Speech 2 [Amo+15], and the unpublished SMLTA system [Gro19]. Deep Speech 1 and 2 are the inspiration for the system implemented during this project. Other related work presented is Google's Listen Attend Spell (LAS) [Cha+15] and IBM's suggested hardware architecture for GPU systems [Dün+18].

The experimental setup is defined based on software setup, workload setup, and hardware setup. The software setup explains how the FE and AM is implemented with TensorFlow. Different inference models referred to as M<number> are defined and how to serve the combined FE, AM, and Beam Search is explained. The served ASR model has a gRPC interface [gRP19] which is later used to serve requests during the experiments. The workload setup uses the Mozilla Common Voice dataset [Moz18], which is identified as continuous speech with independent speakers using a medium sized vocabulary. The dataset is preprocessed by filtering the elements based on frame length and by padding the elements to the same length. The hardware setup is done on two different systems, referred to as RebelRig and Sim. Both systems have four GPUs, which are utilized by duplicating the model to the GPUs and calculating the updated weights on the CPU. A data pipeline where a batch of elements is prepared on the CPU while the GPUs are processing the previous batch is described. This data pipeline enables faster training by utilizing the resources better.

Two different kinds of experiments are conducted in this project. The first is GPU utilization during training and the second is efficiency and quality of the served model. By training different models on a single GPU and measuring throughput, it is concluded that increasing the batch size during training results in more elements processed per second and that larger models have lower throughput. Furthermore, it is concluded that the throughput is better using the PCIe3x16-connected GPU rather than the PCIe2x1-connected GPU. When training on several GPUs, a close to linear relation is found between throughput and number of GPUs. It is further concluded that when training on Sim, the GPUs are fully utilized, hence it is compute-bound whereas RebelRig is memory-bound because of the PCIe2x1 connection.

The experiments on the served model inspect the latency, throughput, and quality of the different models trained during the project. Latency is defined as the time it takes for the client to be serviced by the model. A linear relation between audio length and latency is observed and the average latency is stable when the

number of concurrent users is less than the number of CPU cores in the system. Throughput is defined as the volume processed over time. The conclusion is that when serving the model on the Sim machine, it can handle up to 40 real-time concurrent users using the best performing model.

The quality of the models defines how close the output sentences are to the target sentences. It is measured using Character Error Rate (CER), Word Error Rate (WER), and Bi-Lingual Evaluation Understudy (BLEU). The WERs of the best models implemented during this project are comparable to the WERs reported in related work, although a different dataset is used, hence the WER cannot be compared directly. The character-based models perform worse than the word-based models and the word-based uni-directional LSTM model performs better than the word-based BiLSTM model. Furthermore, different FE methods are experimented with and the Log Mel FE demonstrates better results than Mel and MFCC. The WER of a model with five LSTM layers are lower than the WER of models with fewer layers, but increasing the number of layers to seven does not improve the WER further. By looking at the efficiency in relation to the quality of the models, it is concluded that fewer layers in a model does not result in lower latency and that the WER is not necessarily high for models with low latency. The trade-off between efficiency and quality is observed with M34, which has low latency and moderately higher WER, and M33, which has moderately higher latency and lower WER.

The project has given rise to several ideas for future work. The future work chapter presents ideas of how to improve quality, training efficiency, model serving and experiment measurements. Future work with these ideas has the potential to improve performance of the ASR systems presented in this thesis.

Appendices

A Costs

Part	Name	Price
RebelRig		
GPU's	RTX 2070	4 x 4.200kr
CPU	i7 6700 k	2.350kr
Motherboard	AsRock H110 Pro BTC+	700kr
RAM	Corsair Vengence LPX DDR4-2666 16 GB	719kr
PSU	Corsair HX1200	1.099kr
SSD	512GB SATA SSD	650kr
Risers	Kolink Rendering Mining Kit	3 x 172kr
total	—	22.834kr
		3.412$
SIM		
GPU's	GTX 1080 TI	4 x 7.000kr
CPU	XEON E5-2630V2	2 x 5.000kr
Case, PSU, etc	—	19000kr*
SSD	512GB SATA SSD	650kr
HHD	1TB SATA HDD	300kr
Ram	126 GB Ram	126 * 50kr
total	—	64.250kr
		9.598$

* Talked with the one buying it

Table A.1: Hardware setup cost of individual parts.

B Inference Models

The models used in this project were made in order over time. Some models are extremely close to duplicates but the overall architecture of the system was changed in between some of these similar models. The reason why we keep all of them is to keep the ability to change which model is served without having to change the code behind, and to remember what types of models have been used and tested.

Nr	Model Name	Layers Description
1	inference	1 Convolutions width 5, going from size 13 -128; Batch norm;LSTM 512 static size;LSTM 512 static size; LSTM 512 static size; Relu;
2	inference2	Same as above, but 3 Convolution going straight width 5, going from size 13 - 64 - 128 - 256;.
3	single_LSTM	Single LSTM layer;
4	single_conv_single_LSTM	Same as above, with a single Convolution 13 - 64 and extra Batch norm.
5	three_conv_single_LSTM	BatchNorm; 3 Convolution going straight width 5, going from size 13 - 64 - 128 - 256 with batchNorms; LSTM;
6	three_conv_three_LSTM	BatchNorm; 3 Convolution going straight width 5, going from size 13 - 64 - 128 - 256 with batchNorms; 3 LSTM layers with BatchNorm;
7	single_LSTM_act	Same as model 3 with Relu on each layer
8	single_conv_single_LSTM_act	Same as model 4 with Relu on each layer
9	three_conv_single_LSTM_act	Same as model 5 with Relu on each layer
10	three_conv_three_LSTM_act	Same as model 6 with Relu on each layer
11	deeep_1	Copy of Deep speech 1 [Han+14] architecture with: BatchNorm; Convlution 5 wide, from 13 to 64; 3 Feed forward layers; LSTM (Not bidirectional like the paper but forward);
12	deeep_1_act	Same as model 11 but with Relu activation

Table B.1: Inference models part one.

Nr	Model Name	Layers
13	single_simple_LSTM	Same as model 3.
14	single_output_layer	only the feed forward layer to alphabet size
15	single_conv	Convolution with width 5 going from 13 - 64; BatchNorm;
16	five_LSTM	Five layers of LSTM with batch normalization;
17	three_LSTM	Three layers of LSTM with batch normalization;
18	three_LSTM_norm_act	Three layers of LSTM with Relu activation and batch normalization;
19	conv_stride1_three_LSTM_norm_act	Batchnorm; Convolution width 8 from inputsize to 256; three LSTMs with Relu and Batch normalization;
20	conv_stride1_three_LSTM_norm_act_drop50	Model 19 with Dropout of 0.5 between each layer
21	conv_stride1_three_LSTM_norm_act_drop25	Model 19 with Dropout of 0.25 between each layer
22	conv_stride1_three_LSTM_norm_act_drop10	Model 19 with Dropout of 0.1 between each layer
23	conv_stride1_seven_LSTM_norm_act	Same as model 19 but with seven LSTM layers, as specified in Deep speech 2[Amo+15]
24	conv_stride1_seven_LSTM_norm_act_drop50	Same as model 23 but with 0.5 Dropout between all layers
25	conv_stride1_three_LSTM_norm_act_float16	A mixed posistion model, using 16 bit precision with the same architecture as model 24.
26	conv_stride1_Bi_LSTM_norm_act	Bidirectional LSTM model with three layers of LSTMs
27	conv_stride1_Bi_LSTM_norm_act_drop_70	Same as 26 with dropout set to 70 percent
28	conv_stride1_Bi_LSTM_norm_act_ No_batch_norm_start	Same as 26 with no batch normalization in the starting layer this was later found out to reduce the models training and precision.
29	Bi_directional_LSTM_3	Same as 28 but bidirectional LSTM.
30	Uni_directional_LSTM_3	Same as 18 but with parameters for setting convolution width, stride. At this point in time we also changed the convolutions from using padding to not padding elements.
31	Uni_directional_LSTM_3_Peep_CUDNN	Same as 30 but with activated peephole in the LSTMs, forget gates in the state of the LSTM set to the same value as dropout. This model was also trying out CUDNN editions of RNN, while it did provide a slightly faster training it was not ideal for the saving and serving on CPU, so it was chaged back to LTMS block cells.
32	Uni_directional_LSTM_1_Peep_CUDNN	Same as 31 but with 1 LSTM layers
33	Uni_directional_LSTM_5_Peep_CUDNN	Same as 31 but with 5 LSTM layers
34	Uni_directional_LSTM_7_Peep_CUDNN	Same as 31 but with 7 LSTM layers

Table B.2: Inference models part two.

C | Sim Setup

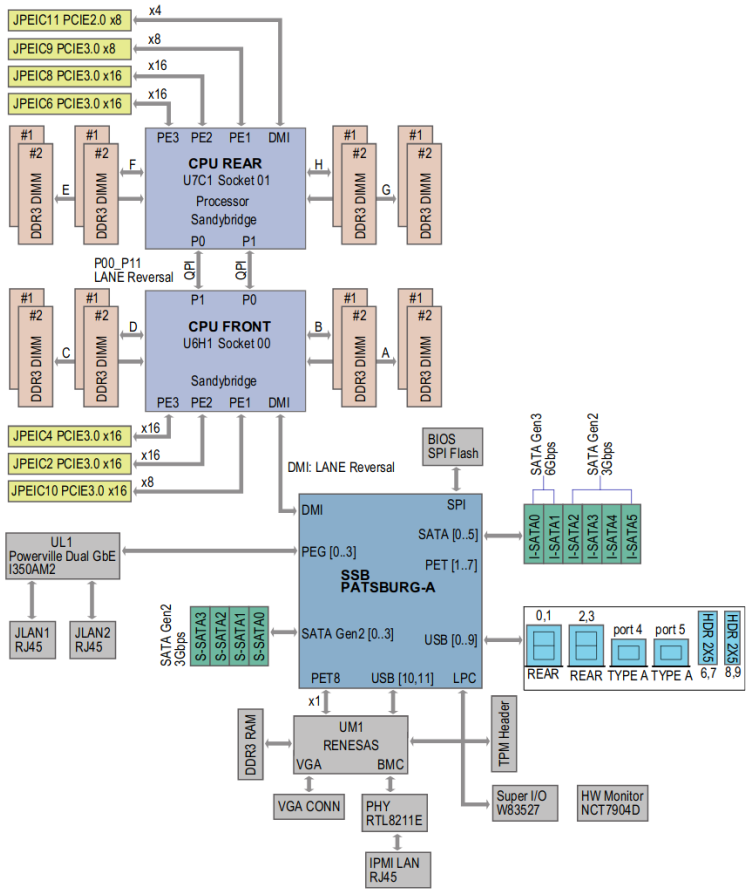


Figure C.1: Sim system hardware architecture [Sup18].

D Inference Model 33

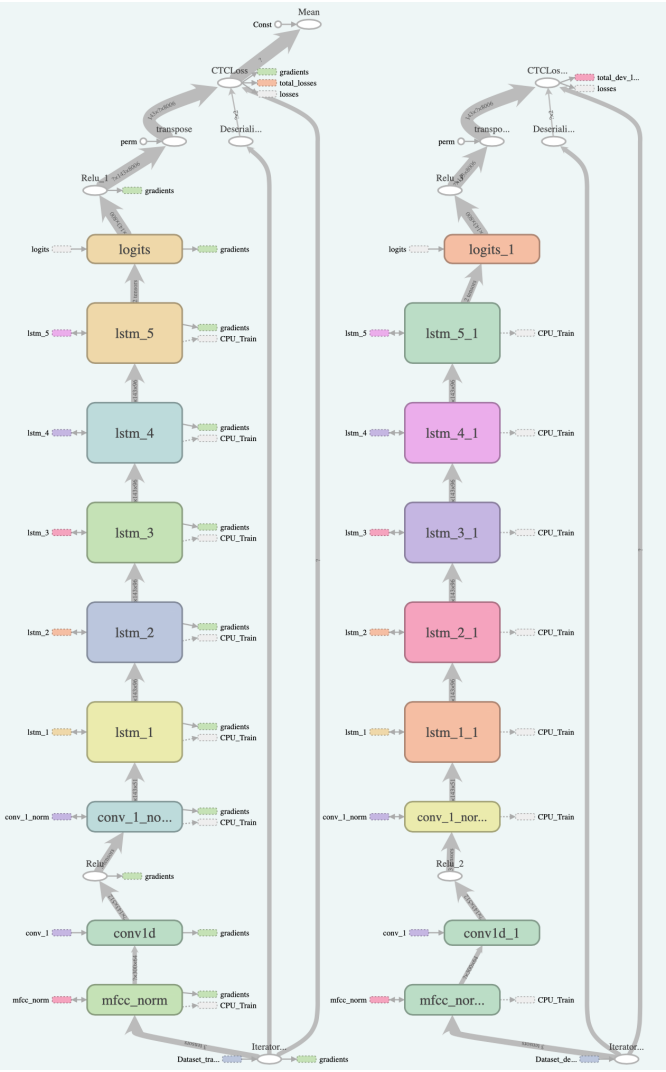


Figure D.1: Overview of AM M33 from TensorBoard.

E Program Arguments

Batch Size: The number of elements per batch during training for each GPU used, meaning that the actual batch size is equal to the number of GPUs used times this argument. The elements are batched using TensorFlow's dataset type.

Iterations: The number of iterations before the training is forcefully terminated. This stopping condition is in addition to the stopping condition based on the validation dataset's loss over the last t iterations of training.

Learning Rate: The learning rate of the optimization algorithm.

Trainer: The trainer/optimizer used for gradient descent. We have used four different trainers for this project AdaDelta, AdaGrad, SGD and Adam as described in chapter 2. The optimization algorithms are numbered from one to four and the number given as argument specifies which optimizer to use.

GPUs: A list of numbers that corresponds to individual GPUs. For instance, if the list $[1,3]$ is given, GPU 1 and GPU 3 are used. The purpose of specifying GPUs is to enable sharing of the GPU cluster with other users and to enable resource scalability experiments.

Number of Files: Specifies how many files from the training dataset are used during training. This was especially useful during initial stages of the project to quickly test whether the system was able to fit to a small part of the training dataset. If the model was not able to converge on a small part of the training dataset, we did not expect it to converge on the entire dataset.

GPU Memory Percentile: The maximum percentage of memory allowed to be used on each GPU. This was also used to share resources with other users but also to benchmark since the default settings of TensorFlow is to allocate all memory.

Model: The inference model used. Each model is referred to by a number.

Network Size: A multiplier that enables scaling of the network size.

Dataset Cutoff Length: The maximum length of an input. This is also used to pad all elements of a size that is lower to the same size.

Dataset Selected: Specifies which type of feature file to load. It is specified by a number that corresponds to a type of feature extraction, for instance MFCC or Mel Spectrograms.

Experiment Name: Specifies the name of the saved model parameters to distinguish between the models of different experiments.

Add Trace: This argument enables tracing of the network to see memory usage and time consumption of different elements.

Stride: The stride of the convolution layer.

Encoding Quality: A parameter that sets the floating point precision of floats to either 32-bit or 16-bit. This is further explained in section 3.4.

Dropout: Dropout rate during training.

Vocabulary: Defines whether to use words or characters as output.

Convolution Width: Specifies the width of the convolution.

Load Checkpoint: The path to a model checkpoint. This is to enable reloading of a model so that if training is interrupted, it can be restarted.

Bibliography

[Ama] Amazon. *Alexa*. URL: https://www.alexa.com/ (cit. on pp. 1, 49).

[Ama19] Amazon. *Machine Learning on AWS*. 2019. URL: https://aws.amazon.com/machine-learning/ (cit. on p. 22).

[Amo+15] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin". In: *CoRR* abs/1512.02595 (2015). arXiv: 1512.02595. URL: http://arxiv.org/abs/1512.02595 (cit. on pp. 1, 4, 12, 13, 20, 21, 28–31, 40, 43–45, 47–50, 55).

[App] Apple. *Siri*. URL: https://www.apple.com/siri/ (cit. on p. 1).

[AsR19] AsRock. *AsRock H110 Pro BTC+ Motherboard*. 2019. URL: https://www.asrock.com/mb/Intel/H110%20Pro%20BTC+/index.asp (cit. on p. 30).

[Bah+16] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. "End-to-end attention-based large vocabulary speech recognition". In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2016, pp. 4945–4949. DOI: 10.1109/ICASSP.2016.7472618 (cit. on pp. 26, 40).

[Bas+19] M. K. Baskar, L. Burget, S. Watanabe, M. Karafiát, T. Hori, and J. H. ernocký. "Promising Accurate Prefix Boosting for Sequence-to-sequence ASR". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2019, pp. 5646–5650. DOI: 10.1109/ICASSP.2019.8682782 (cit. on p. 43).

[Bat+17] Eric Battenberg, Jitong Chen, Rewon Child, Adam Coates, Yashesh Gaur, Yi Li, Hairong Liu, Sanjeev Satheesh, David Seetapun, Anuroop Sriram, and Zhenyao Zhu. "Exploring Neural Transducers for End-to-End Speech Recognition". In: *CoRR* abs/1707.07413 (2017). arXiv: 1707.07413. URL: http://arxiv.org/abs/1707.07413 (cit. on pp. 4, 13, 20, 48, 49).

[Bur+11] Phil Burk, Larry Polansky, Douglas Repetto, Mary Robers, and Dan Rockmore. *Music And Compters, A Theoretical and Historical Approach*. 2011. URL: http://sites.music.columbia.edu/cmc/MusicAndComputers/ (cit. on pp. 3, 4).

[CC14] Boxing Chen and Colin Cherry. "A Systematic Comparison of Smoothing Techniques for Sentence-Level BLEU". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, June 2014, pp. 362–367. DOI: 10.3115/v1/W14-3346. URL: https://www.aclweb.org/anthology/W14-3346 (cit. on pp. III, 41, 42).

[Cha+15] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. "Listen, Attend and Spell". In: *CoRR* abs/1508.01211 (2015). arXiv: 1508.01211. URL: http://arxiv.org/abs/1508.01211 (cit. on pp. 1, 4, 21, 29, 43, 45, 47, 49, 50).

[Chi+17] Chung-Cheng Chiu, Tara N. Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J. Weiss, Kanishka Rao, Katya Gonina, Navdeep Jaitly, Bo Li, Jan Chorowski, and Michiel Bacchiani. "State-of-the-art Speech Recognition With Sequence-to-Sequence Models". In: *CoRR* abs/1712.01769 (2017). arXiv: 1712.01769. URL: http://arxiv.org/abs/1712.01769 (cit. on p. 1).

[CMU] CMUSphinx. *Basic concepts of speech recognition*. URL: https://cmusphinx.github.io/wiki/tutorialconcepts/ (cit. on p. 6).

[Coa+13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. "Deep learning with COTS HPC systems". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1337–1345. URL: http://proceedings.mlr.press/v28/coates13.html (cit. on pp. 21, 22).

[CPH14] G. Chen, C. Parada, and G. Heigold. "Small-footprint keyword spotting using deep neural networks". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2014, pp. 4087–4091. DOI: 10.1109/ICASSP.2014.6854370 (cit. on p. 49).

[Dea+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. "Large Scale Distributed Deep Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1223–1231. URL: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf (cit. on pp. 22, 36).

[Dün+18] Celestine Dünner, Thomas P. Parnell, Dimitrios Sarigiannis, Nikolas Ioannou, Andreea Anghel, and Haralampos Pozidis. "Snap ML: A Hierarchical Framework for Machine Learning". In: *CoRR* abs/1803.06333 (2018). arXiv: 1803.06333. URL: http://arxiv.org/abs/1803.06333 (cit. on pp. 22, 50).

[Ess19] Essentia. *Essentia MonoLoader*. 2019. URL: https://essentia.upf.edu/documentation/reference/streaming_MonoLoader.html (cit. on p. 24).

[Fis19] Tim Fisher. *PCI Express (PCIe)*. Jan. 2019. URL: https://www.lifewire.com/pci-express-pcie-2625962 (cit. on p. 22).

[Gal15] Mitch Gallagher. *7 Questions about sample rate*. 2015. URL: https://www.sweetwater.com/insync/7-things-about-sample-rate/ (cit. on p. 3).

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*. 2010 (cit. on pp. 7, 8).

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Page numbers does not match the book. Cambridge, MA, USA: MIT Press, 2016. ISBN: 9780262035613. URL: http://www.deeplearningbook.org (cit. on pp. III, 7–18, 21).

[GH17] Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017. ISBN: 9781627052986 (cit. on pp. 9, 10).

[Gol16] Yoav Goldberg. "A Primer on Neural Network Models for Natural Language Processing". In: *J. Artif. Int. Res.* 57.1 (Sept. 2016), pp. 345–420. ISSN: 1076-9757. URL: http://dl.acm.org/citation.cfm?id=3176748.3176757 (cit. on pp. 9, 10).

[Goo] Google. *Google Assistant*. URL: https://assistant.google.com/ (cit. on p. 1).

[Goo19a] Google. *Cloud AI Google*. 2019. URL: https://cloud.google.com/products/ai/ (cit. on p. 22).

[Goo19b] Google. *Google Compute Engine Pricing*. 2019. URL: https://cloud.google.com/compute/pricing#gpus (cit. on p. 22).

[Goo19c] Google. *Protocol Buffers*. 2019. URL: https://developers.google.com/protocol-buffers/ (cit. on pp. 19, 28).

[Gra+06] 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*. ICML '06. Pittsburgh, Pennsylvania, USA: ACM, 2006, pp. 369–376. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143891. URL: http://doi.acm.org/10.1145/1143844.1143891 (cit. on p. 13).

[Gre13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, 2013. ISBN: 9780133390094 (cit. on pp. 33, 38, 39).

[Gre17] Brendan Gregg. Aug. 2017. URL: http://www.brendangregg.com/usemethod.html (cit. on p. 33).

[Gro19] Baidu Research Group. *A Breakthrough in Speech Technology: Baidu Launched SMLTA, the First Streaming Multi-layer Truncated Attention Model for Large-scale Online Speech Recognition*. Jan. 2019. URL: http://research.baidu.com/Blog/index-view?id=109 (cit. on pp. 1, 20, 21, 38, 48, 50).

[gRP19] gRPC. *gRPC: A high performance, open-source universal RPC framework*. 2019. URL: https://grpc.io/ (cit. on pp. 19, 50).

[GSS03] Felix A. Gers, Nicol N. Schraudolph, and Jürgen Schmidhuber. "Learning Precise Timing with Lstm Recurrent Networks". In: *J. Mach. Learn. Res.* 3 (Mar. 2003), pp. 115–143. ISSN: 1532-4435. DOI: 10.1162/153244303768966139. URL: https://doi.org/10.1162/153244303768966139 (cit. on p. 11).

[GSW19] J. Guo, T. N. Sainath, and R. J. Weiss. "A Spelling Correction Model for End-to-end Speech Recognition". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2019, pp. 5651–5655. DOI: 10.1109/ICASSP.2019.8683745 (cit. on pp. 4, 21, 43, 49).

[Han+14] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. "Deep Speech: Scaling up end-to-end speech recognition". In: *CoRR* abs/1412.5567 (2014). arXiv: 1412.5567. URL: http://arxiv.org/abs/1412.5567 (cit. on pp. 1, 4, 20, 21, 36, 43, 47, 49, 50, 54).

[Has19] Masroor Hasan. *How we improved Tensorflow Serving performance by over 70%*. Mar. 2019. URL: https://hackernoon.com/how-we-improved-tensorflow-serving-performance-by-over-70-f21b5dad2d98 (cit. on p. 48).

[HDA11] X. He, L. Deng, and A. Acero. "Why word error rate is not a good metric for speech recognizer training for the speech translation task?" In: *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2011, pp. 5632–5635. DOI: 10.1109/ICASSP.2011.5947637 (cit. on p. 41).

[HHS17] Elad Hoffer, Itay Hubara, and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 1731–1741. URL: http://papers.nips.cc/paper/6770-train-longer-generalize-better-closing-the-generalization-gap-in-large-batch-training-of-neural-networks.pdf (cit. on p. 15).

[IBM18] IBM. *IBM Power System AC922, An acceæration superhighway to the AI era*. 2018. URL: https://www.ibm.com/dk-en/marketplace/power-systems-ac922 (cit. on p. 23).

[IBM19] IBM. *Watson IBM Cloud AI*. 2019. URL: https://www.ibm.com/cloud/ai (cit. on p. 22).

[ICA] IEEE ICASSP. *2019 IEEE International Conference on Acoustics, Speech and Signal Processing*. URL: https://2019.ieeeicassp.org/ (cit. on pp. 21, 48).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (cit. on pp. III, 14, 15).

[ITU88] ITU-T. *General Aspects of Digital Transmission Systems, ITU-T Recommendation G.711*. 1988. URL: https://www.itu.int/rec/T-REC-G.711-198811-I/en (cit. on p. 3).

[JH13] Navdeep Jaitly and Geoffrey E. Hinton. "Vocal Tract Length Perturbation (VTLP) improves speech recognition". In: 2013 (cit. on pp. 17, 18).

[Jia+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014) (cit. on p. 30).

[JM09] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009. ISBN: 0131873210 (cit. on pp. 5, 29).

[Joh+14] Maree Johnson, Samuel Lapkin, Vanessa Long, Paula Sanchez, Hanna Suominen, Jim Basilakis, and Linda Dawson. "A systematic review of speech recognition technology in health care". In: *BMC Medical Informatics and Decision Making* 14.1 (Oct. 2014), p. 94. ISSN: 1472-6947. DOI: 10.1186/1472-6947-14-94. URL: https://doi.org/10.1186/1472-6947-14-94 (cit. on p. 1).

[Jun06] Andrew Jungwirth. *Beam Search Algorithm*. 2006. URL: http://jhave.org/algorithms/graphs/beamsearch/beamsearch.shtml (cit. on p. 13).

[Kar19] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. 2019. URL: https://cs231n.github.io/convolutional-networks/ (cit. on pp. 8, 9).

[Kes+16] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836 (cit. on p. 14).

[Kin18] Jason Kincaid. *Challenges in Measuring Automatic Transcription Accuracty*. 2018 (cit. on p. 40).

[Kip+17] Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems." In: *EDBT*. 2017, pp. 49–60 (cit. on p. 49).

[Koe16] Philipp Koehn. *Neural Networks Language Models*. Apr. 2016 (cit. on p. 7).

[Kri10] Alex Krizhevsky. *Convolutional deep belief networks on cifar-10*. 2010 (cit. on p. 12).

[Kri14] Alex Krizhevsky. "One weird trick for parallelizing convolutional neural networks". In: *CoRR* abs/1404.5997 (2014). arXiv: 1404.5997. URL: http://arxiv.org/abs/1404.5997 (cit. on pp. 21, 22, 36).

[KS16] Peter Knees and Markus Schedl. *Music Similarity and Retrieval: An Introduction to Audio- and Web-based Strategies (The Information Retrieval Series)*. Springer, 2016. ISBN: 3662497204 (cit. on pp. 3–5).

[KSE12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386 (cit. on pp. 21, 22).

[Lyo17] Richard F. Lyon. *Human and Machine Hearing: Extracting Meaning from Sound*. Cambridge University Press, 2017. DOI: 10.1017/9781139051699 (cit. on pp. 4, 5).

[Mar+15] Martin 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, Dan Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". In: (2015). Software available from tensorflow.org. URL: http://tensorflow.org/ (cit. on p. 18).

[MHN13] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. 2013 (cit. on p. 12).

[Mic] Microsoft. *Cortana*. URL: https://www.microsoft.com/en-us/cortana (cit. on p. 1).

[Mic+17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. "Mixed Precision Training". In: *CoRR* abs/1710.03740 (2017). arXiv: 1710.03740. URL: http://arxiv.org/abs/1710.03740 (cit. on pp. 31, 32, 48).

[Mic19] Microsoft. *AI Platform Microsoft Azure*. 2019. URL: https://azure.microsoft.com/en-us/overview/ai-platform/ (cit. on p. 22).

[Mis+05] Gilad Mishne, David Carmel, Ron Hoory, Alexey Roytman, and Aya Soffer. "Automatic Analysis of Call-center Conversations". In: *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. CIKM '05. Bremen, Germany: ACM, 2005, pp. 453–459. ISBN: 1-59593-140-6. DOI: 10.1145/1099554.1099684. URL: http://doi.acm.org/10.1145/1099554.1099684 (cit. on p. 1).

[Mor18] Timothy Pricket Morgan. May 2018. URL: https://www.nextplatform.com/2018/05/09/ibm-rounds-out-power9-systems-for-hpc-analytics/ (cit. on p. 23).

[Moz18] Mozilla. *Common Voice*. 2018. URL: https://voice.mozilla.org/en (cit. on pp. 21, 28, 29, 44, 50).

[Moz19] Mozilla. *A TensorFlow implementation of Baidu's DeepSpeech architecture*. 2019. URL: https://github.com/mozilla/DeepSpeech (cit. on p. 21).

[Nav01] Gonzalo Navarro. "A Guided Tour to Approximate String Matching". In: *ACM Comput. Surv.* 33.1 (Mar. 2001), pp. 31–88. ISSN: 0360-0300. DOI: 10.1145/375360.375365. URL: http://doi.acm.org/10.1145/375360.375365 (cit. on p. 40).

[NVI12] NVIDIA. *NVML API Reference Manual*. Oct. 2012. URL: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVML_cuda5/nvml.4.304.55.pdf (cit. on p. 33).

[NVI16] NVIDIA. "NVIDIA Tesla P 100". In: (2016). URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (cit. on p. 32).

[NVI17] NVIDIA. *Nvlink Fabric, Advancing Multi-GPU Processing*. 2017. URL: https://www.nvidia.com/en-us/data-center/nvlink/ (cit. on p. 23).

[NVI18] NVIDIA. "NVIDIA Turing GPU Architecture". In: (2018). URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (cit. on pp. 32, 48).

[NVI19a] NVIDIA. *Tensor Cores*. 2019. URL: https://www.nvidia.com/en-us/data-center/tensorcore/ (cit. on p. 32).

[NVI19b] NVIDIA. *Training with Mixed Precision*. Apr. 2019. URL: https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html (cit. on pp. III, 31–33).

[Ors19] Gram Orsinium. *Text Distance*. 2019. URL: https://github.com/orsinium/textdistance (cit. on p. 40).

[Pap+02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://www.aclweb.org/anthology/P02-1040 (cit. on p. 41).

[Per19] Eric Perbos-Crinck. *RTX 2060 Vs GTX 1080Ti Deep Learning Benchmarks: Cheapest RTX card Vs Most Expensive GTX card*. Feb. 2019. URL: https://towardsdatascience.com/rtx-2060-vs-gtx-1080ti-in-deep-learning-gpu-benchmarks-cheapest-rtx-vs-most-expensive-gtx-card-cd47cd9931d2 (cit. on pp. 32, 48).

[PN07] Maja Popovic and Hermann. Ney. "Word error rates: Decomposition over POS classes and applications for error analysis." In: (2007), pp. 48–55 (cit. on p. 40).

[Pro18] NLTK Project. *NLTK Translate Package*. Nov. 2018. URL: https://www.nltk.org/api/nltk.translate.html (cit. on p. 42).

[PyT] PyTorch. *Multi-GPU Examples*. URL: https://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html# (cit. on p. 30).

[Rab89] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *PROCEEDINGS OF THE IEEE*. 1989, pp. 257–286 (cit. on p. 6).

[Raf+17] Colin Raffel, Thang Luong, Peter J. Liu, Ron J. Weiss, and Douglas Eck. "Online and Linear-Time Attention by Enforcing Monotonic Alignments". In: *CoRR* abs/1704.00784 (2017). arXiv: 1704.00784. URL: http://arxiv.org/abs/1704.00784 (cit. on pp. 1, 43).

[RJ86] L. R. Rabiner and B. H. Juang. "An introduction to hidden Markov models". In: *IEEE ASSp Magazine* (1986) (cit. on p. 6).

[RMN09] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. "Large-scale Deep Unsupervised Learning Using Graphics Processors". In: ICML '09 (2009), pp. 873–880. DOI: 10.1145/1553374.1553486. URL: http://doi.acm.org/10.1145/1553374.1553486 (cit. on pp. 1, 21).

[Ros17] Adrian Rosebrock. *How-To: Multi-GPU training with Keras, Python, and deep learning*. Oct. 2017. URL: https://www.pyimagesearch.com/2017/10/30/how-to-multi-gpu-training-with-keras-python-and-deep-learning/ (cit. on p. 30).

[Ros18] Frank Rosner. *Window Functions in Stream Analytics*. Oct. 2018. URL: https://dev.to/frosnerd/window-functions-in-stream-analytics-1m6c (cit. on p. 49).

[Sai+15] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak. "Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Apr. 2015, pp. 4580–4584. DOI: 10.1109/ICASSP.2015.7178838 (cit. on p. 5).

[SBS05] D. Steinkraus, I. Buck, and P. Y. Simard. "Using GPUs for machine learning algorithms". In: *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. Aug. 2005, 1115–1120 Vol. 2. DOI: 10.1109/ICDAR.2005.251 (cit. on pp. 1, 21).

[Sch19] Harald Scheidl. *Github: CTC Word Beam Search Decoding Algorithm*. 2019. URL: https://github.com/githubharald/CTCWordBeamSearch (cit. on p. 47).

[SFS18] H. Scheidl, S. Fiel, and R. Sablatnig. "Word Beam Search: A Connectionist Temporal Classification Decoding Algorithm". In: *16th International Conference on Frontiers in Handwriting Recognition*. IEEE. 2018, pp. 253–258. DOI: 10.1109/ICFHR-2018.2018.00052 (cit. on p. 47).

[Shi+18] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. "Graph Processing on GPUs: A Survey". In: *ACM Comput. Surv.* 50.6 (Jan. 2018), 81:1–81:35. ISSN: 0360-0300. DOI: 10.1145/3128571. URL: http://doi.acm.org/10.1145/3128571 (cit. on p. 13).

[Smi02] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. 2002. URL: https://www.dspguide.com/ch6/2.htm (cit. on p. 8).

[Spe15] Lucia Specia. *Statistical Machine Translation*. July 2015 (cit. on p. 7).

[Sri+14] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958. URL: http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf (cit. on p. 18).

[Sri+17] Anuroop Sriram, Heewoo Jun, Sanjeev Satheesh, and Adam Coates. "Cold Fusion: Training Seq2Seq Models Together with Language Models". In: *CoRR* abs/1708.06426 (2017). arXiv: 1708.06426. URL: http://arxiv.org/abs/1708.06426 (cit. on pp. 1, 40).

[SSB14] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition". In: *CoRR* abs/1402.1128 (2014). arXiv: 1402.1128. URL: http://arxiv.org/abs/1402.1128 (cit. on p. 11).

[Ste13] James Stewart. *Fourier Series*. 2013. URL: https://www.stewartcalculus.com/data/CALCULUS%20Early%20Transcendentals/upfiles/FourierSeries5ET.pdf (cit. on p. 5).

[Str15] Nikko Strom. "Scalable distributed DNN training using commodity GPU cloud computing". In: *INTERSPEECH*. 2015 (cit. on pp. 33, 36).

[Sup18] SuperMicro. *X9DRG-QF*. Oct. 2018. URL: https://www.supermicro.com/products/motherboard/Xeon/C600/X9DRG-QF.cfm (cit. on pp. 30, 56).

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011. ISBN: 978-0-321-57351-3 (cit. on p. 13).

[Ten18] TensorFlow. *TensorFlow Serving Batching Guide*. Dec. 2018. URL: https://github.com/tensorflow/serving/blob/master/tensorflow_serving/batching/README.md (cit. on p. 48).

[Ten19a] TensorFlow. 2019. URL: https://www.tensorflow.org/lite (cit. on pp. 1, 49).

[Ten19b] TensorFlow. *Architecture*. 2019. URL: https://www.tensorflow.org/tfx/serving/architecture (cit. on pp. 1, 28).

[Ten19c] TensorFlow. *Data Input Pipeline Performance*. 2019. URL: https://www.tensorflow.org/guide/performance/datasets (cit. on pp. 31, 32).

[Ten19d] TensorFlow. *Docker*. 2019. URL: https://www.tensorflow.org/install/docker (cit. on p. 27).

[Ten19e] TensorFlow. *Introduction*. 2019. URL: https://www.tensorflow.org/guide/low_level_intro (cit. on p. 19).

[Ten19f] TensorFlow. *Simple Audio Recognition*. 2019. URL: https://www.tensorflow.org/tutorials/sequences/audio_recognition (cit. on p. 18).

[Ten19g] TensorFlow. *TensorBoard: Visualizing Learning*. 2019. URL: https://www.tensorflow.org/guide/summaries_and_tensorboard (cit. on p. 19).

[Ten19h] TensorFlow. *TensorFlow: Advanced Convolutional Neural Networks*. 2019. URL: https://www.tensorflow.org/tutorials/images/deep_cnn (cit. on pp. 30, 31).

[Ten19i] TensorFlow. *TensorFlow Architecture*. 2019. URL: https://www.tensorflow.org/guide/extend/architecture (cit. on p. 19).

[Ten19j] TensorFlow. *TensorFlow Serving with Docker*. 2019. URL: https://www.tensorflow.org/tfx/serving/docker (cit. on pp. 27, 28).

[Ten19k] TensorFlow. *TensorFlow tf.nn.ctc_loss*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/ctc_loss (cit. on p. 26).

[Ten19l] TensorFlow. *TensorFlow tf.signal.mfccs_from_log_mel_spectrograms*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/signal/mfccs_from_log_mel_spectrograms (cit. on p. 24).

[Ten19m] TensorFlow. *TensorFlow tf.train.AdadeltaOptimizer*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/train/AdadeltaOptimizer (cit. on p. 26).

[Ten19n] TensorFlow. *TensorFlow tf.train.AdagradOptimizer*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/train/AdagradOptimizer (cit. on p. 26).

[Ten19o] TensorFlow. *TensorFlow tf.train.AdamOptimizer*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer (cit. on p. 26).

[Ten19p] TensorFlow. *TensorFlow: Using GPUs*. 2019. URL: https://www.tensorflow.org/guide/using_gpu (cit. on p. 30).

[Ten19q] TensorFlow. *tf.contrib.layers.xavier_initializer*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer (cit. on pp. 8, 26).

[Ten19r] TensorFlow. *tf.contrib.rnn.LSTMBlockCell*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/contrib/rnn/LSTMBlockCell (cit. on p. 27).

[Ten19s] TensorFlow. *tf.layers.batch_normalization*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/layers/batch_normalization (cit. on pp. 27, 32).

[Ten19t] TensorFlow. *tf.layers.dense*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/layers/dense (cit. on p. 27).

[Ten19u] TensorFlow. *tf.nn.bidirectional_dynamic_rnn*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/bidirectional_dynamic_rnn (cit. on p. 27).

[Ten19v] TensorFlow. *tf.nn.ctc_beam_search_decoder*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/ctc_beam_search_decoder (cit. on p. 28).

[Ten19w] TensorFlow. *tf.nn.dynamic_rnn*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn (cit. on p. 27).

[Ten19x] TensorFlow. *tf.nn.rnn_cell.LSTMCell*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/rnn_cell/LSTMCell (cit. on p. 11).

[Ten19y] TensorFlow. *tf.nn.rnn_cell.relu6*. 2019. URL: https://www.tensorflow.org/api_docs/python/tf/nn/relu6 (cit. on p. 12).

[Tri17] Robert Triggs. *Why noise cancellation is the most important tech in mobile right now*. Dec. 2017. URL: https://www.androidauthority.com/noise-cancellation-explained-820149/ (cit. on p. 3).

[Vae19] Nik Vaessen. *Word Error Rate for Automatic Speech Recognition*. 2019. URL: https://pypi.org/project/jiwer/ (cit. on p. 41).

[Var17] Rohan Varma. *Who introduced GPU to deep learning*. May 2017. URL: https://www.quora.com/Who-introduced-GPU-to-deep-learning (cit. on p. 21).

[VD12] C Vimala and V Dr. Radha. "A Review on Speech Recognition Challenges and Approaches". In: (2012) (cit. on pp. 3, 40).

[WB18] Sebastian Benjamin Wrede and Sebastian Baunsgaard. "Thesis Preparation: Scalable Stream-Based Automatic Speech Recognition". IT University of Copenhagen. Dec. 2018 (cit. on p. 3).

[Wil18] Cody Marie Wild. *Convolution: An Exploration of a Familiar Operators Deeper Roots*. Oct. 2018. URL: https://towardsdatascience.com/convolution-a-journey-through-a-familiar-operators-deeper-roots-2e3311f23379 (cit. on p. 8).

[YD15] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach (Signals and Communication Technology)*. Springer, 2015. URL: https://www.springer.com/us/book/9781447157786 (cit. on pp. 1, 4–7, 24, 40).

[Zei12] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701 (cit. on pp. III, 16, 17).

[Zha+] Jian Zhang, Risheng Xia, Zhonghua Fu, Junfeng Li, and Yonghon Yan. *A fast two-microphone noise reduction algorithm based on power level ratio for mobile phone* (cit. on p. 3).

[ZK18] Giancarlo Zaccone and Md. Rezaul Karim. *Deep Learning with TensorFlow - Second Edition: Explore Neural Networks and Build Intelligent Systems with Python*. 2nd. Packt Publishing, 2018. ISBN: 9781788831109 (cit. on pp. 1, 49).