Large Scale Speech Recognition with Deep Learning

Anand Umashankar

Aalto University
School of Science

Technische Universität Berlin
Faculty IV, Institute of Software Engineering

Master's Programme in ICT Innovation - Data Science

Master's Thesis
Espoo, September 29, 2021

Supervisors: Prof. Mikko Kurimo, Aalto University
Prof. Sebastian Möller, TU Berlin
Advisor: Aku Rouhe M.Sc.

Aalto University
Technische Universität Berlin
Master's Programme in ICT Innovation - Data Science

ABSTRACT OF
MASTER'S THESIS

Author:	Anand Umashankar		
Title:	Large Scale Speech Recognition with Deep Learning		
Date:	September 29, 2021	**Pages:**	64+19
Major:	Data Science	**Code:**	SCI3095
Minor:	Innovation & Entrepreneurship	**Language:**	English
Aalto Student Number: 765837			
TU Berlin Marticulation Number: 0453065			
Supervisors:	Prof. Mikko Kurimo Prof. Sebastian Möller		
Advisor:	Aku Rouhe M.Sc.		
Keywords:	Speech Recognition, Deep Learning, Large-scale, Multi-GPU training, Hogwild, Data Parallel, Attention Models, Recurrent Neural Networks, SpeechBrain, PyTorch		

Abstract:
Automatic Speech Recognition (ASR) is the task of converting speech signal into text. To enable usage of large datasets to train the end to end speech recognition neural networks, we build a pipeline for increasing efficiency in data storing, data loading and training. We train an attention based sequence-to-sequence model and use word error rates for evaluating the experiments. The time to reach benchmark accuracy is another important metric used to compare the training efficiency of different systems. This work uses a dataset with around 26,000 hours that is new for speech to text experiments. The dataset consists of conference calls with a diverse set of speakers. Around half of the data has a presentation style audio, while the other half contains conversational language.

First, the work focuses on the steps taken to make this dataset efficient for speech recognition. Then, two types of distributed training algorithms, synchronous and asynchronous training, are applied which enables the usage of multiple GPUs for stochastic gradient descent. The comparison of the different methods show that, for the experiment setup employed in this work, synchronous training provides the best word error rate of 10.87%. This run converged in 32 hours using 4 GPUs in parallel, which is a speed-up of 2x compared to the single GPU training job to reach the benchmark word error rate. The effective batch size plays an important role in these results. The experiment results also show that increasing the scale of the data, reduces the overall training time, and hence using larger datasets is beneficial even when obtaining quicker training results is an important criterion.

Aalto University
Technische Universität Berlin
Master's Programme in ICT Innovation - Data Science Zusammenfassung

Autor:	Anand Umashankar		
Titel:	Large Scale Speech Recognition with Deep Learning		
Datum:	September 29, 2021	**Seiten:**	64+19
Haupt:	Datenwissenschaft	**Koodi:**	SCI3095
Unerheblich:	Innovation & Unternehmertum	**Sprache:**	Englisch
Aalto Student Number: 765837			
TU Berlin Marticulation Number: 0453065			
Vorgesetzte:	Prof. Mikko Kurimo Prof. Sebastian Möller		
Berater:	Aku Rouhe M.Sc.		

Abstract:
Automatische Spracherkennung (ASR) ist die Aufgabe der Umwandlung von Sprachsignalen in Text. Um die Verwendung großer Datensätze für das Training neuronaler Netze für die durchgängige Spracherkennung zu ermöglichen, bauen wir eine Pipeline zur Steigerung der Effizienz bei der Datenspeicherung, dem Laden von Daten und dem Training. Wir trainieren ein aufmerksamkeitsbasiertes Sequenz-zu-Sequenz-Modell und verwenden Wortfehlerraten zur Auswertung der Experimente. Die Zeit bis zum Erreichen der Benchmark-Genauigkeit ist eine weitere wichtige Kennzahl, um die Trainingseffizienz verschiedener Systeme zu vergleichen. In dieser Arbeit wird ein Datensatz mit rund 26.000 Stunden verwendet, der für Experimente mit Sprache zu Text neu ist. Der Datensatz besteht aus Konferenzgesprächen mit einer Vielzahl von Sprechern. Etwa die Hälfte des Datensatzes besteht aus Audio im Präsentationsstil, während die andere Hälfte Konversationssprache enthält. Die Arbeit konzentriert sich zunächst auf die Schritte, die unternommen wurden, um diesen Datensatz für die Spracherkennung effizient zu machen. Dann werden zwei Arten von verteilten Trainingsalgorithmen, synchrones und asynchrones Training, angewandt, welche die Verwendung mehrerer GPUs zum stochastischen Gradientenabstieg ermöglichen. Der Vergleich der verschiedenen Methoden zeigt, dass für den in dieser Arbeit verwendeten Versuchsaufbau das synchrone Training die beste Wortfehlerrate von 10,87% liefert. Dieser Durchlauf konvergierte in 32 Stunden unter Verwendung von 4 GPUs parallel, was eine Geschwindigkeitssteigerung um das Zweifache im Vergleich zu einem einzelnen GPU-Trainingsjob bedeutet, um die Benchmark-Wortfehlerrate zu erreichen. Die Ergebnisse des Experiments zeigen auch, dass eine Erhöhung der Datenmenge die Gesamttrainingszeit verkürzt und daher die Verwendung größerer Datensätze von Vorteil ist, selbst wenn die Erzielung schnellerer Trainingsergebnisse ein wichtiges Kriterium ist.

ನ ಹಿ ಜ್ಞಾನೇನ ಸದೃಶಂ

na hi jnanena sadrusham
(Nothing is equal to knowledge)

- Bhagavad Gita 4.38

Acknowledgements

I will forever be grateful to my family and friends who have supported me through thick and thin.

I have loved working on this topic and I thank my supervisor, Prof. Mikko Kurimo, for giving me the opportunity and steering me in the right direction throughout my thesis. I have enjoyed working with Aku Rouhe, whose calm and knowledgeable feedback has driven most of the work in this thesis. I have looked to meetings with Aku, whenever I felt confused, and I thank him for providing some much-needed inspiration. I wish that the pandemic would have allowed us to have more offline interactions.

The work was done in the Aalto ASR group, and I would like to thank every group member. I thank the Aalto Scientific Computing group for the help with using the cluster computation resources. They have been helpful and friendly when solving my issues with Triton on their garage sessions. I thank Prof. Jukka Sihvonen, for providing me the access to the dataset, without which this work would not have existed. I thank Prof. Sebastian Möller for remotely supervising this work.

Finally, I thank you, the reader, for taking time to glance through this work. I hope you enjoy it!

TUB Declaration - Section 46 (8)

I hereby declare that I have carried out this work independently and without any unauthorized help and using only the sources and tools cited.

September 30, 2021

(Anand Umashankar)

Contents

List of Figures

List of Tables

Acronyms

AED Attention-based Encoder Decoder.

ASR Automatic Speech Recognition.

BRNN Bidirectional Recurrent Neural Networks.

BSP Bulk Synchronous Parallel.

CER Character Error Rate.

CER-P Character Error Rate with Punctuations.

CNN Convolutional Neural Networks.

CRDNN Convolutional Recurrent Deep Neural Networks.

CTC Connectionist Temporal Classification.

DDP Distributed Data-Parallel.

DNN Deep Neural Networks.

ESPNet End-to-end Speech Processing Toolkit.

GPU Graphics Processing Unit.

GRU Gated Recurrent Unit.

HMM Hidden Markov Model.

HTK Hidden Markov Model Toolkit.

LSTM Long Short-Term Memory.

MLP Multilayer Perceptron.

RNN Recurrent Neural Network.

SGA Sequential Gradient Accumulation.

SGD Stochastic Gradient Descent.

WER Word Error Rate.

WER-P Word Error Rate with Punctuations.

Chapter 1

Introduction

Automatic Speech Recognition (ASR) is the process of converting speech into text. ASR is used in a wide range of applications like digital assistants, captioning audio conversations, etc. ASR systems convert digital speech audio to text transcriptions using pattern matching techniques based on similarity of features computed from the audio and match it with the models which are trained from data. Research in this field has become very dynamic in the previous decade with the application of deep neural networks for ASR. Deep Neural Networks (DNN) are a subset of machine learning models, which consist of multiple layers of computation units and learn to provide the required output for the given input. Applying deep neural networks for ASR requires speech audio data along with transcripts. This is then fed to the deep neural networks to help it to learn from the audio data, and this process is called model training. State-of-the-art ASR systems that use deep learning require large amounts of training data.

Increasing the data used for training deep neural network models exposes more diverse data to the models, which have previously led to substantial increases in accuracy. Moreover, considering the decreasing costs of data storage and network costs[56], scaling up training to higher and higher amounts of training data has become an interesting trend over the recent years. Popular cloud-based solutions offered by big enterprises use the vast magnitude of resources available to them to train deep learning models that can process speech audio with high levels of accuracy[32]. These large-scale experiments are harder to conduct for smaller companies and for academic institutions, because of the need for high amounts of storage and processing power for the process of training, which are not easily available. Hence, there is a need for optimizing the training process to reduce the storage footprint, processing power, and resources required to train large-scale deep learning models.

Large dataset training jobs generally use distributed training, which means that multiple processing units are used with the data present in a network location. This introduces new problems like optimizing input/output read performance and managing local storage, etc. These are the problems regarding infrastructure

and engineering related domains. Apart from these, there are also research-based problems which include convergence issues when using stochastic gradient descent in a distributed setup of training.

1.1 Problem statement

The purpose of this thesis is to explore the task of large-scale speech recognition and to optimize the training jobs which enables more wide scale adaptation of scaling up datasets using for training end to end ASR models. The aim of the thesis, therefore, are as follows:

1. How to enable large-scale training tasks from an engineering point of view? What are the best methods to store data, load data for such training jobs?
2. What is the effect of training data scale on the quality of the acoustic model used?
   (a) Does increasing the scale of training data affect the convergence time of the acoustic model?
3. How effective are the multi-GPUs and multiple processing jobs to accelerate the training time and reach convergence quicker?
   (a) Compare synchronous and asynchronous training methods and analyse which of them are suitable for our system setup?
   (b) What is the speed-up achieved by using these methods for training an acoustic end to end model?

See Section 6.1 for the direct answers to these problem statements.

1.2 Structure of the Thesis

Chapter 2 discusses the most essential concepts in deep learning and the network architectures used for the end to end ASR model. Chapter 3 is more specific to large-scale speech recognition and its importance. It also discusses the datasets available for large-scale speech recognition. Chapter 4 explains the dataset used, and the techniques used to scale up the speech recognition task. Next, we delve into the experiments conducted and show the results in Chapter 5. Chapter 6 answers the research problems and talks about the future work for the thesis. Finally, Chapter 7 concludes the thesis with the most important takeaways from the work.

Chapter 2

Background

2.1 Overview

This chapter focuses on the fundamental concepts for the rest of the chapters. We begin with introducing hybrid speech recognition systems and then compare them with end to end speech recognition systems in Section 2.2. Then we discuss the different types of deep neural networks used in an ASR application. Finally, we discuss the SpeechBrain toolkit in Section 2.4, which is used for all of our experiments.

2.2 Speech Recognition

2.2.1 Hybrid speech recognition systems

Hybrid method of speech recognition use multiple steps which include feature generation, acoustic modelling and language modelling which combine to form a system which provides speech to text results. Figure 2.1 shows this architecture.

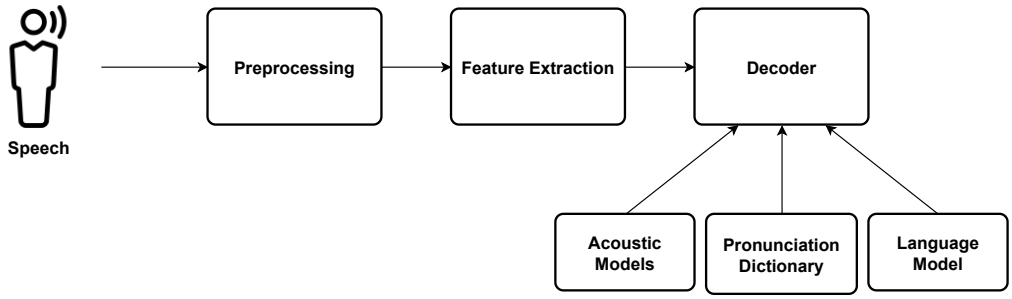

Figure 2.1: A hybrid ASR system architecture

In the HMM-based architecture, the first step is preprocessing the speech signal to generate features which are robust to noise and convert the signal to vectors which are used in the decoder [40]. The information of how different words are pronounced is stored in a pronunciation dictionary, also known as a lexicon. The goal of the decoder is finding the optimal word sequence, $\hat{\boldsymbol{W}} = W_1, W_2, \ldots, W_m$ with the number of words, m, is not known. This can be mathematically defined as maximizing the posterior probability $P(\boldsymbol{W} \mid \boldsymbol{O})$ [35].

$$\begin{aligned} \hat{\boldsymbol{W}} &= \arg\max_{\boldsymbol{W}} P(\boldsymbol{W} \mid \boldsymbol{O}) \\ &= \arg\max_{\boldsymbol{W}} \frac{P(\boldsymbol{O} \mid \boldsymbol{W})P(\boldsymbol{W})}{P(\boldsymbol{O})} \\ &= \arg\max_{\boldsymbol{W}} P(\boldsymbol{O} \mid \boldsymbol{W})P(\boldsymbol{W}) \end{aligned} \tag{2.1}$$

Applying the Bayes rule on, $P(\boldsymbol{W} \mid \boldsymbol{O})$ we end up with the Equation 2.1, which is a combination of two probability models. $P(\boldsymbol{O} \mid \boldsymbol{W})$ corresponds to the acoustic model, which computes the likelihood of the observed speech signal, given a possible word sequence and $P(\boldsymbol{W})$ corresponds to the language model [43]. The language model represents our prior beliefs about what sequences of words are more or less likely. We say ”prior” because this is knowledge that we have before we even process any speech signal.

2.2.2 End to end speech recognition systems

In end to end speech recognition systems, a single model, usually based on deep learning, replaces the hybrid ASR system stages. A deep learning model combined with an external language model can achieve higher performance than hybrid speech pipelines for particular applications [14]. These systems rely on large neural networks, and as the size of the network grows and has millions of parameters, they require larger amounts of speech data to be trained well. Smaller models with lesser number of parameters can be trained with smaller amounts of data, but may fail to generalize well[58]. Since the system learns the whole task end-to-end, specialized components for capturing finer details of the speaker or other noise filtering components are not essential. On the contrary, previous experiments have shown that end to end models stand out in the cases where robustness is crucial[23].

2.3 What are Deep Neural Networks?

A deep neural network is a neural network with many hidden layers. Deep neural networks can be trained to model non-linear models and functions for very high dimensional data [22]. Traditionally, it was quite challenging to train deep neural

networks, but there was a resurgence in the usage of deep neural networks in the previous decade, also in the field of speech recognition [10, 13, 23, 39].

One of the first deep neural network was a Multilayer Perceptron (MLP), which consists of an input layer, an output layer and a hidden layer. Figure 2.2 shows this architecture.

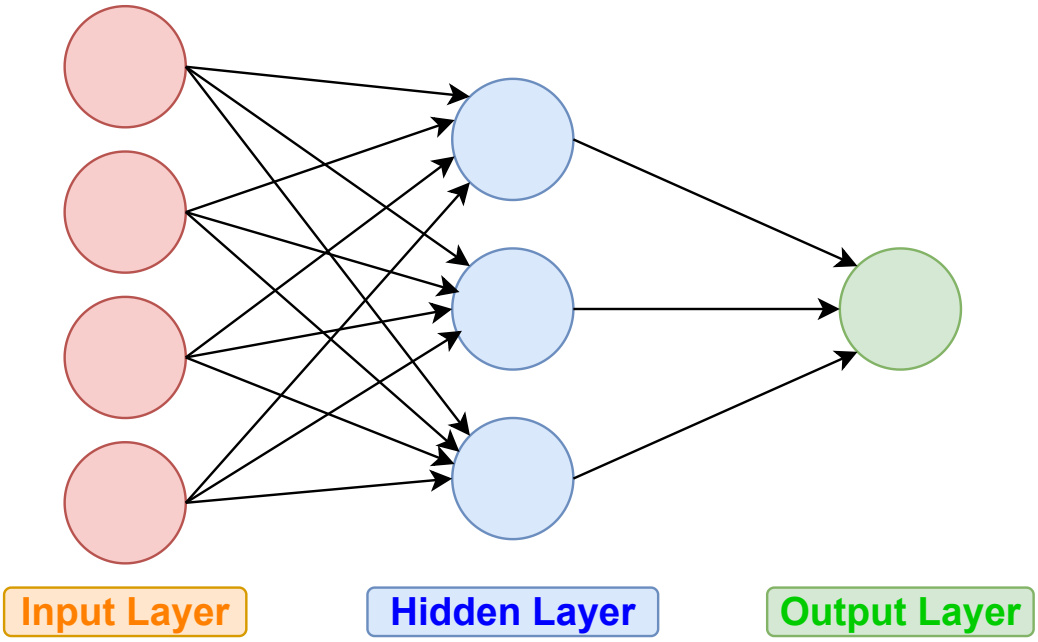

Figure 2.2: An MLP architecture

Each layer of this network consists of multiple neurons, and each of these neurons can be represented using a vector, and all the neuronal connections in a layer can then be represented using a weights matrix that determines how much a neuron should take part in the decision-making process. At each layer, multiply the input signal with its corresponding weights and then add them up to obtain a weighed sum as the output of a layer. This can be represented as,

$$z = w_1x_1 + w_2x_2 + .. + w_nx_n$$

In the above formula z is the weighted sum output, w is the weight, x is an input and there are n number of inputs. An activation function like tanh, sigmoid, ReLU, etc is applied to the weighted sum to get the output of the layer.

2.3.1 Model training

Training of deep learning models use labelled data, split to train, validation and test splits. The model gradually learns to predict the labels of training data by

adjusting the model weights based on the error. The validation split of the data is used to measure the accuracy of unseen data, periodically, during the training process. The test split, also unseen by the model, is not used during the process of training and used for evaluation of the different models after training them. The most important metric studied is on the test data split [28].

Let w be a vector of model weights, and $x(w)$ be a loss function such that when given y, the ground truth and the predicted label x, measures the difference between them. Model training is an optimization problem that finds the best w which minimizes the average loss over training data. In practice, this is accomplished using Stochastic Gradient Descent (SGD) [55], an iterative algorithm that adapts w using a few training samples at a time.

$$\mathrm{w}_{n+1} = \mathrm{w}_n - \gamma_n \nabla \ell_{\mathrm{B}_n}(\mathrm{w}_n) \tag{2.2}$$

where γ_n is the learning rate in the n-th iteration of the algorithm, B_n is a mini-batch of b training samples, and $\nabla \ell$ is the gradient of the loss function, averaged over the batch samples:

$$\nabla \ell_{\mathrm{B}_n}(\mathrm{w}_n) = \frac{1}{b} \sum_{\mathrm{x} \in \mathrm{B}_n} \nabla \ell_{\mathrm{x}}(\mathrm{w}_n)$$

The whole training process hence happens in two stages. First, a batch of labelled data propagates forward through the model to compute the output. This is compared to the ground-truth, measuring the loss value. Then, the error propagates backwards from the last layer to the first layer. This error is used to compute the gradient for the weights in each layer, and the weights can then be updated incrementally by applying Equation 2.2.

When training such a deep learning model, the aim is to reduce time taken to achieve convergence with a target test accuracy results (time-to-benchmark). There are two factors which can affect the time-to-accuracy [28].

1. **Statistical Efficiency:** The number of iterations that a training algorithm such as SGD requires to find a solution with a given test accuracy. This can be done by optimizing the model architecture and improving the training data.

2. **Hardware Efficiency:** The execution time of each iteration. This can be done by improving and making maximum use of the computational infrastructure.

The focus of this thesis is hence to optimize the training by increasing the scale of data (statistical efficiency), while making maximum use of the available computational resources (hardware efficiency).

Next, we look at the deep neural network architectures that are used in this work.

2.3.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of neural network that can predict a future state by taking the previous states as input [19]. This network can be thought of as having memory because it considers the context of the input provided to determine the output state [21].

Figure 2.3 represents the recurrent unit of an RNN. It consists of an extra synapse that loops back into itself. In practise, this means that each unit receives a new input and also receives the output from the previous units as an input.

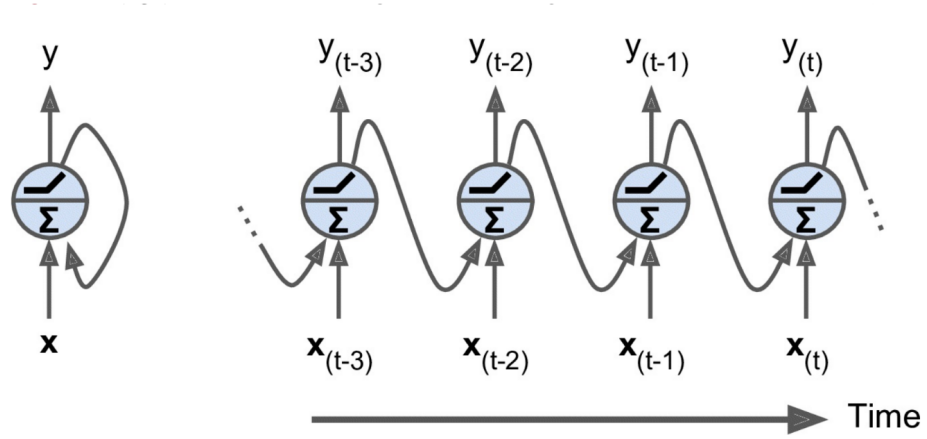

Figure 2.3: A basic recurrent neural network cell and a way to visualize it unrolling through time, from time step t-3 to t [21]

Consider an input series $\boldsymbol{x} = (x_1, \ldots, x_T)$, an RNN calculates the hidden sequence $\boldsymbol{h} = (h_1, \ldots, h_T)$ and the output sequence $\boldsymbol{y} = (y_1, \ldots, y_T)$ by looping the following equations from $t = 1$ to T :

$$h_t = \mathcal{H}\left(w_{xh}x_t + w_{hh}h_{t-1} + b_h\right)$$
$$y_t = w_{hy}h_t + b_y$$

where the w terms denote weight matrices, the b terms denote bias vectors and $\mathcal{H}$ is the hidden layer function [19].

The drawback of an RNN is that they use only the previous output to get the current context, but for speech recognition there is value to also utilize the future states too. A more advanced variation of RNN is hence a Bidirectional Recurrent Neural Networks (BRNN) [57], which processes the input in both directions using two hidden layers and then this used as input to a single output layer.

2.3.2.1 Long Short-term Memory (LSTM)

The main drawback with vanilla RNNs is the vanishing gradient problem, which happens because the hidden states pass through a series of `tanh` activation functions, which leads to gradients becoming very close to zero and the network stops training through backpropagation [25]. A Long Short-Term Memory (LSTM) cell [25] modifies the RNN architecture by passing another state vector to the next time step, along with the hidden state. The cell state variable enables the useful long term-based information to be preserved because this is modified only when there is new information. To control the cell state, gates which multiply a signal that is a continuous value ranging between 0 and 1 are used. Each gate has a weight matrix which are trained by iterating over the input data. The original work use two gates which are "input" gate, which control what information is essential and hence adds to the current state, and "output" gate which control what information needs to be written to the hidden state. More recently, a third gate called a "forget" gate controls deleting unwanted information from the cell state to stop cells growing in a boundless manner [18].

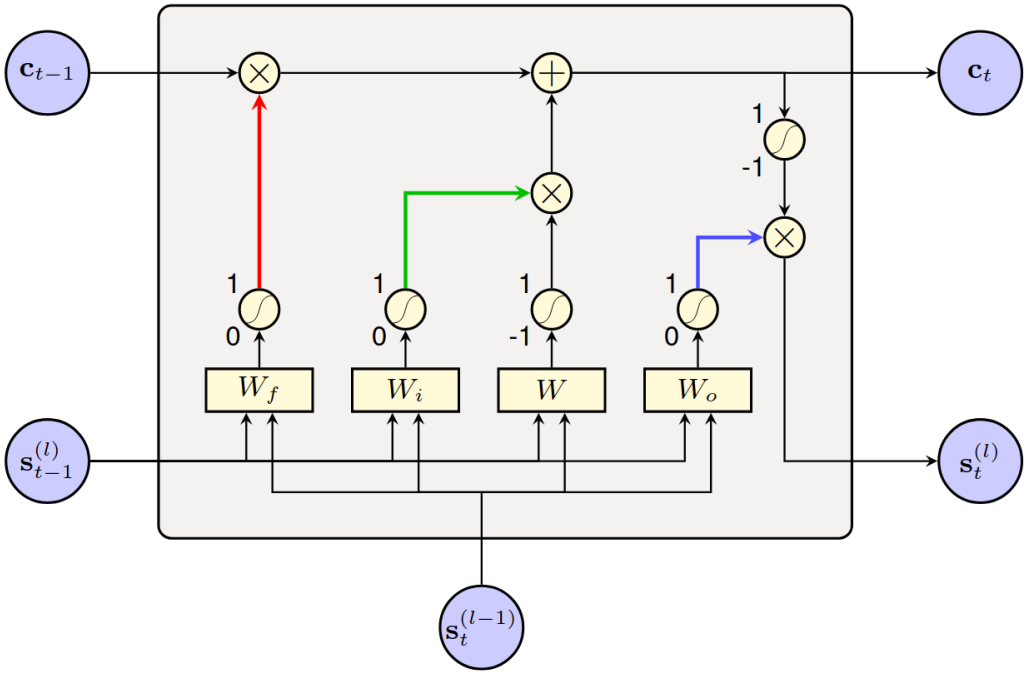

Figure 2.4: A basic LSTM cell for a single time step [16]

Figure 2.4 shows the architecture of an LSTM layer with all the three gates. The three gates take in identical input, which also depends on the definition of the network. In most cases, the input to LSTM cell is the combination of the hidden

state of the previous time step (c_{t-1}) and the output of the previous layer. Each gate contains 2 weight matrices, one each for the two inputs, and a bias vector. (Figure 2.4 shows only a weight matrix per gate.)

2.3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a branch of deep neural networks, which are hierarchical in nature and rely on convolution operations to process input. This helps to process data for tasks which can use spatial information by sharing the weights across the inputs. Hence, CNNs are especially powerful for performing tasks like image classification, object detection, etc on images and videos. CNNs perform well when spatial features are important to perform the given task [30].

The architecture of a general CNN is alternating layers of convolution with subsampling layers [8]. Figure 2.5 shows the architecture of AlexNet, one of the most popular CNNs for image classification [30]. The architecture consists of convolutional layers altered with max pooling layers which act as sub sampling layers. The last layer of the network is using a fully connected layer to reduce the final dimensionality to match the required output shape.

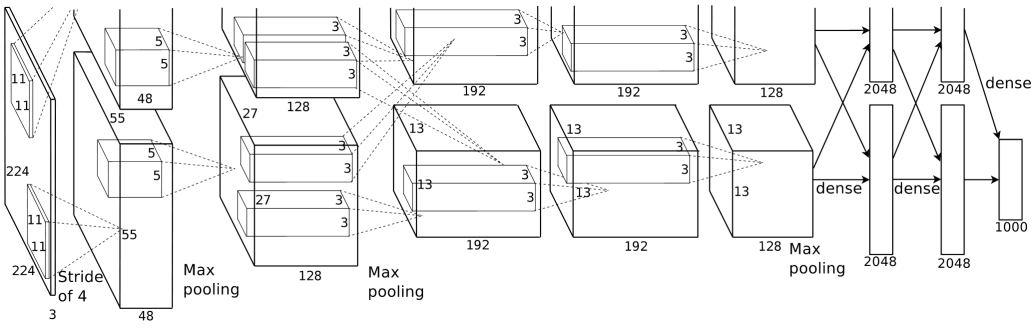

Figure 2.5: AlexNet, a popular CNN architecture [30]

More recently, convolutional neural networks have become popular for end to end speech recognition applications as well [67]. The idea is to use convolutional layers along with a recurrent neural network. Unlike a typical RNN with a fully connected layer, these networks replace it with a convolution layer because it is better at using the input topology to produce the output. These neural networks which are a combination of convolutional layers and a recurrent network are hence termed as Convolutional Recurrent Deep Neural Networks (CRDNN).

2.3.4 Atteniton-based Encoder Decoder (AED)

Encoder-decoder type architecture models have become extremely popular over the last few years, and one of the variations in this branch of neural networks are the

attention-based models [51]. Attention-based Encoder Decoder (AED) models are a combination of two submodules: the Encoder module and the Decoder module. The encoder is an acoustic model which converts the input signal to an embedding representation. The decoder module takes the embeddings generated to produce a probability distribution for the output character sequences [4, 67].

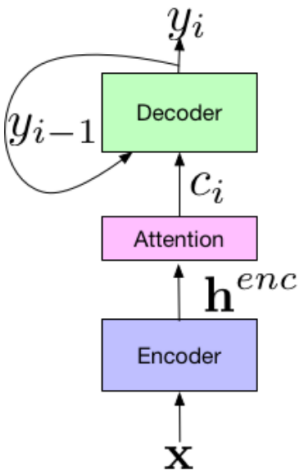

Figure 2.6: Architecture Diagram for a AED model [6]

Figure 2.6 represents the architecture of a AED model with the encoder, attention, and decoder modules. The attention mechanism provides the context value. The below equations represents the functions in mathematical form.

$$\mathbf{h} = \text{Encoder}(\mathbf{x})$$
$$P\left(y_i \mid \mathbf{x}, y_{<i}\right) = \text{Decoder}\left(y_{<i}, \mathbf{h}\right)$$

where $\mathbf{h}$ is the embedding representation, the output from the encoder module and $P\left(y_i \mid \mathbf{x}, y_{<i}\right)$ represents the probability of the output character based on the input signal and the previous output characters.

Let $\mathbf{x} = (x_1, \ldots, x_T)$ be the input sequence and y be the output sequence of characters. AED models produce for every character an output y_i which is a conditional distribution based on the previously occurring sequence $y_{<i}$ and the original signal x by making use of the chain rule for probabilities:

$$P(\mathbf{y} \mid \mathbf{x}) = \prod_i P\left(y_i \mid \mathbf{x}, y_{<i}\right)$$

This defines the end to end nature of the AED models because it generates a probability distribution for the output characters sequence right from the input signal.

AED models output character scores for each sequence, and since datasets usually do not have character level alignments for speech data and their transcription. Training speech recognizers can become harder to train, than what it initially seems. To solve this issue, we use Connectionist Temporal Classification (CTC) loss to train AED model.

2.3.4.1 Connectionist Temporal Classification (CTC) Loss

The CTC criteria is a way of training end-to-end models when the target sequence and the input sequence do not match exactly in length. It eliminates the need to align the target labels on a frame to frame basis with the input training signal. When processing a speech that consists of the word "to", since the rate of speech varies from person to person, it is hard for the model to know the difference between "toooooo" or "to". We have to remove all duplicate "o"s. But this will not work if the actual text required is "too"? Then removing all duplicate "o"s gets us the wrong result. CTC loss helps to manage this.

From the previous encoder-decoder architecture, the difference is that the conditional probability of the output character at each time step, the encoder output embeddings $\mathbf{h}$, are then fed to a softmax layer which considers the whole set of blank-augmented output symbols to predict a probability distribution output similar in nature to the typical encode decoder output.

2.4 SpeechBrain Toolkit

Speech recognition has always been driven forward due to the critical role played by open-source toolkits such as HTK [64], Kaldi [50], etc. More recently, general purpose deep learning frameworks like TensorFlow [1] and PyTorch [48] have become useful for speech recognition tasks and this has led to newer toolkits like ESPNet [62]. SpeechBrain [53] is a toolkit designed to be flexible and used for multiple tasks to speed up research and development of speech related technologies. SpeechBrain is a PyTorch-based toolkit and equipped to perform multiple tasks at once like recognize speech, understand its content, emotion, language, and speakers.

2.4.1 Brain class

The SpeechBrain's core functionalities revolve around the `Brain` class, which defines a general training loop. The `Brain.fit()` method trains the models, and the Figure 2.7 shows the basic components of the method.

Training of most DNN models can be completed with a few lines of code. The Brain class also handles repetitive boilerplate code. It also calls the validation,

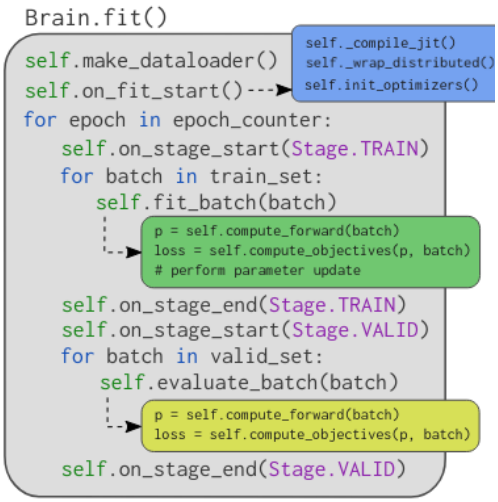

Figure 2.7: `Brain.fit()` method illustration [53]

learning rate scheduling, recoverable fault tolerant checkpointing modules. Training a model happens through a python script and run from the command line with the combination of a configuration file:

```
python train.py train.yaml
```

2.4.2 HyperPyYAML

The configuration file is in YAML format which is human-readable and can be used to define the model architecture, the data files, the parameters of the model, hyperparameters for training and other features of the pipeline. SpeechBrain utilities HyperPyYAML [1] which makes several extensions to the default YAML file format. The most important addition out of them is the easier object creation. An example:

```
epoch_counter: !new:speechbrain.utils.epoch_loop.EpochCounter
    limit: 100
```

This tag with prefix `!new:` creates an instance of the specified class with an option to pass keyword arguments by using a mapping node like in the example above.

[1] speechbrain/HyperPyYAML: Extensions to YAML syntax for better python interaction. https://github.com/speechbrain/HyperPyYAML

The next extension is the use of prefix, `!name` which simplifies the creation of an interface for specifying a function or class or other static Python entity. Below is an example using that.

```
opt_class: !name:torch.optim.Adam
    lr: !ref <lr>
```

The prefix `!ref` is also an extension to the alias system used in YAML files. It takes keys in angles brackets and searched for the key within that YAML file. It can be used also for string interpolation, concatenation etc. It also has the `!include` prefix to import other YAML files to be used to reference in the current YAML file. This can be used to make the configuration files modular as well.

```
dataset_parameters: !include:dataset.yaml
tokenizer_parameters: !include:tokenizer.yaml
```

The last extension is the use of tuples. implicitly resolve any string starting with `(` and ending with `)` to a tuple.

2.4.3 Other features

SpeechBrain extends the PyTorch's data loading to help with speech data related challenges like handling variable length sequences and complicated data transformations. By leveraging PyTorch's `Dataset` and `DataLoader` classes, SpeechBrain enables on-the-fly data transformation (loading, augmentation, environment corruption, text processing, text encoding, feature extraction, etc) and can also be scaled to multiple parallel workers.

SpeechBrain also supports dynamic batching and multi GPU training, which are discussed in detail in Chapter 4. This enables large-scale speech recognition tasks, which is the core of the work done in this thesis.

SpeechBrain provides recipes for some of the most popular deep learning models and processing units which are used for different datasets. These can be used as is for the designed tasks, or they can also be modified to be used for custom tasks. This makes the development easier and initial bottleneck to start building ASR models is reduced.

Chapter 3

Large-Scale Speech Recognition

3.1 Overview

End to end speech recognition models have millions of parameters and the architecture complexities make them highly data hungry [32]. Modern ASR systems have been designed to work in multiple domain and environment conditions, and this robustness is possible due to the usage of larger and larger datasets. One of the largest datasets is the 300,000 hours dataset from Google's research team. In their work, they have attempted to build a domain invariant speech recognition model using the large dataset[41]. Other applications include multi-lingual ASR models [27] and highly accurate domain-specific ASR models. These experiments showcase the ability of the ASR systems to perform at high levels by using more and more data. An observation from almost all of these works is that the researches are limited to an industrial domain and published by Google, Microsoft, etc who go through fewer budget constraints than other researchers in the domain of ASR. In the academic domain, the amount of research done in this direction is limited due to the resource constrains concerning GPU resource, CPU resources, network communication availability of datasets, disk storage.

3.2 Scaling up training

One of the major factors that have led to the rise in popularity of DNNs, is due to the increase in the scale of data used to train them. There are three main dimensions in which this can be done. First is the magnitude of training data. The model performance can be improved by feeding more data to the deep neural network during training [24]. The definition of a *large-scale* dataset specific to speech recognition is ever-changing. For some context, a popular research in 2014 [23], used 5000 hours of training data and more recent research in 2020 [42], researchers have used 300,000 hours of training data with random augmentation

techniques for each epoch. Hence, we can observe that, In the previous 7 years, the training data used in researches has grown by multiple folds. Increasing the amount of high-quality training data hence is one of the unequivocal ways to improve performance of deep learning models. Hence, the practice of adding more and more training data is likely to continue through the next few years [36].

The second dimension is the scale of the infrastructure. The easily available parallel hardware, especially graphics processing units (GPUs), has proven to be a great enabler to train DNNs in shorter times than before [66]. This is also due to the reducing costs of hardware resources. Storing and handling data have become cheaper and easier [56]. GPU resources have become cheaper and more powerful, which have enabled researchers to use more data for training of deep neural networks [56].

The third dimension is the size of the DNN models, by increasing the width and depth of the models, DNN models have increased in architectural complexity to achieve better results [11].

3.3 Distributed Training

Complex models trained on large datasets have shown good results [11]. Such huge training tasks can take models a few weeks or even months on a single GPU to reach convergence. To increase the throughput of the training system, one of the straightforward methods is by increasing the amount of compute resources, which include the number of GPUs available. Distributed training is the method of training in a distributed infrastructure of multiple compute nodes, each with multiple GPUs on it [31].

This also brings with it many challenges. The first challenge is to use the computing resources efficiently. This should also go along with tight integration of hardware elements to improve the overall throughput of the system. Data transmissions across machines can be a common bottleneck when training a network. During training a deep neural network using SGD, the weights have to be synchronized across all the devices in use. As the amount of GPUs in the system grows, the data transmission overhead also grows with it. These challenges require research at a confluence of both computing systems and deep learning training methods, and is receiving growing attention [5, 9, 34, 49, 63].

3.3.1 Types of parallelism

There are many ways to train deep learning models in parallel. The most common ones are discussed here, namely data and model parallelism.

3.3.1.1 Model Parallelism

In model parallelism, split the deep neural network into different devices and load a portion of the model on each device for training, as shown in Figure 3.1. The device that holds the first layer receives the training data. Next, the data propagates through the rest of the forward pass on the device, and it passes the output to the device that holds the next layer of the DL model. During the backward pass, compute the gradients starting from the device that hold the output layer and then propagate the gradients to the devices backward till the device with the input layer.

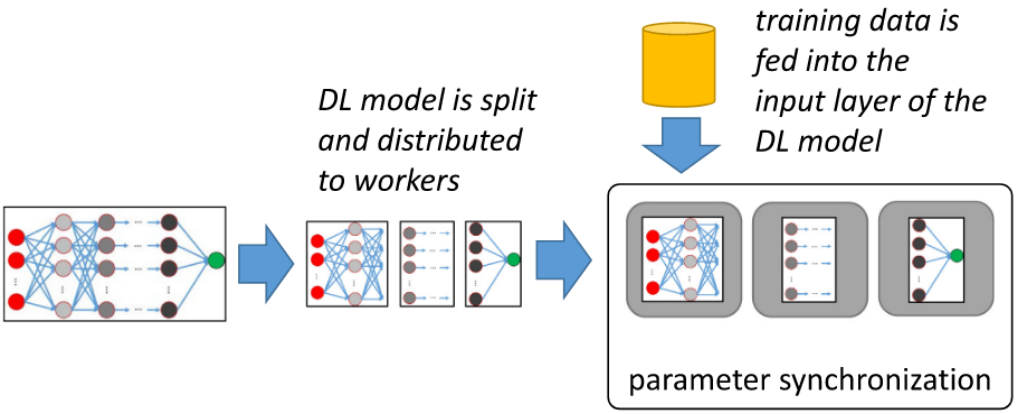

Figure 3.1: Architecture Diagram of Model Parallelism [36]

The biggest challenge in model parallelism is to split the model into partitions such that the hardware efficiency on different devices is high during training [37]. In many cases, the best way is to empirically try out various permutations and measure the performance and keep the best permutation that has worked well. The second drawback is that model parallelism needs heavy communication between devices. A combined effect of these two challenges is that stalling may occur if models are hard to be split effectively to reduce the communication overloads and synchronization delays. Therefore, training speed might not increase linearly by increasing the number of parallel devices [38].

The major benefit of the model parallelism is that it can accommodate huge deep learning networks, which cannot be stored on a single device (GPU) to train them because the memory required to train them is split across multiple units.

3.3.1.2 Data Parallelism

In data parallelism, copy the deep neural network into different devices and load an identical copy of the model on each device for training, as shown in Figure 3.2. Split the data into the same number as the number of devices, and then data passes

into the model replicas of the workers for training. Perform the training process on each chunk of the data that is assigned to the device to generate partial gradient updates of the model parameters. Once it is completed on all the processing units, the parameters are synchronized across all the devices [28].

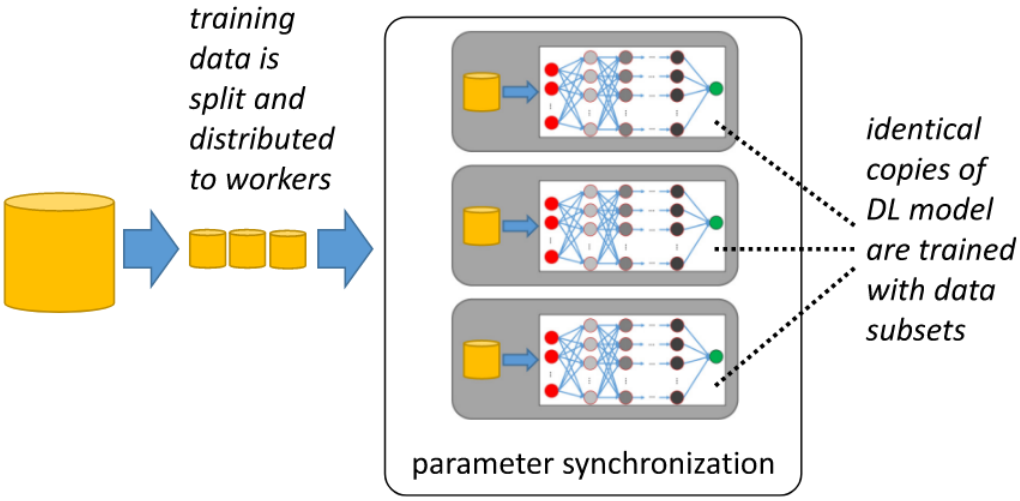

Figure 3.2: Architecture Diagram of Data Parallelism [36]

Since each device processes a mini batch of data, the total batch size is the product of the mini batch with the number of GPUs. This high data scale might lead to poor convergence. The other drawback is that ideally, the data that is split along the different workers needs to be identically distributed, so that the parameter updates by the different workers can be easily synchronized to get the overall model [26].

The major benefit of data parallelism is that it can be easily applied to most of the deep learning networks without much specific knowledge about the architecture of the model. It is highly scalable for the cases when the training is compute intensive but have relatively few parameters like RNNs, CNNs [29].

3.3.2 Types of synchronization

Due to the multiple nodes trying to update the model, the important question then becomes when to synchronize the parameters between the parallel workers. The main trade off is between managing training convergence and quality with synchronization cost to update all the models on the parallel workers. There are mainly two methods which are discussed here: synchronous and asynchronous methods.

3.3.2.1 Synchronous

The workers synchronize the weight updates after every batch of data processed. This is implemented in some of the first and prominent methods, like the Bulk Synchronous Parallel (BSP) [59] method, which is already available in popular data analytics platforms. The main advantage is that the model convergence is easier due to the strict nature of synchronization. However, this method is vulnerable to a situation where *one* single worker stalls the entire training process, especially when the different nodes in the system are heterogeneous in nature [7].

Synchronous training is already available in many open-source DL frameworks, such as TensorFlow [1] and PyTorch [48]. Most ideally, it is best suited for a system where the nodes involved are homogeneous in nature with similar hardware configurations and hence stragglers are not a significant issue and there are minimal delays from communication.

3.3.2.2 Asynchronous

The workers update the model parameters completely independent of other workers. Asynchronous training relies on the theoretical characteristic of SGD that the model convergence is robust to the random ordering of data. The model used by a worker at some point of time might be a stale model because there are no binding guarantees. This makes it difficult to assess model convergence using asynchronous training. However, the advantage is that the workers enjoy great levels of flexibility during training, hence avoiding the straggler problem that is faced in the synchronous method.

Hogwild [45] is one implementation of the asynchronous training of parallel SGD. Hogwild works by providing all the workers access to a shared memory space, which consists of the model parameters. This is completely lock-free, and hence seems dangerous because new model parameters from one worker can be completely overwritten by another worker without being used at all. Nevertheless, it has been shown that that since model updates are anyway sparse in nature when done sequentially, Hogwild also can achieve results similar to sequential training. This has been applied to large neural networks with good results [15].

3.3.3 Data Parallel Systems Architecture

The architecture of the system defines *how* the parameter synchronization among different workers happens in a data parallel system. There are three major considerations for the system architecture. Firstly, the architecture must be able to scale up to many parallel workers to update the DL model and also read the updated DL model. Secondly, the system needs to be configurable, that is, to be able to manage different parameter settings, etc. The third aspect is to be compatible with widely available inter process communication primitives.

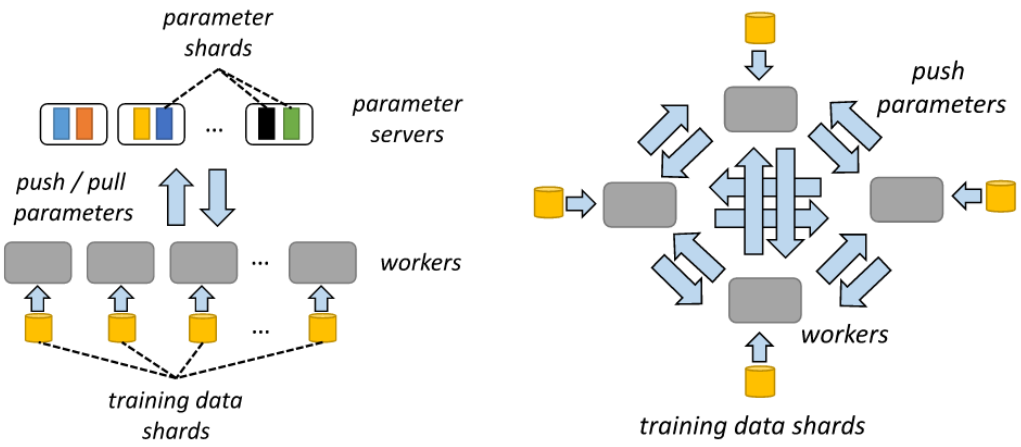

Figure 3.3: Diagram of centralized architecture on the left and decentralized architecture on the right. [36]

3.3.3.1 Centralized: Parameter Server

Centralized architecture consists of a parameter server which is located (logically) in the centre and the worker nodes updates their computed gradients to the central server and all the workers then read the updated model to update their replicas of the model. This architecture is one of the most commonly used in a data parallel system. It is also common to use parameter shards to accelerate the process of communicating the parameter updates. Figure 3.3 shows this architecture.

3.3.3.2 Decentralized: allreduce Operation

Decentralized architecture operates without a Parameter Server. The parameters synchronize directly by using an `allreduce` operation as shown in Figure 3.3.

The major benefit of using a decentralized architecture is that there is no need for one node reserved for parameter server. There is also no single point of failure, and this makes it easier to manage fault tolerance. The major drawback of the decentralized architecture is that communication increases exponentially with the increase in number of workers. One way to avoid this is to use partial gradient updates. For each synchronization, every worker sends a partition of the gradients to every other worker. The communication costs now depend on the number of partitions *and* also on the number of workers in the system.

Both of these architectures, are widely adopted in leading open-source deep learning frameworks, however the decentralized architecture is the most suitable architecture for a system where there is a single node with multiple GPUs attached because communication on the same node does not consume too much of resources. However, for multinode training, a centralized architecture works better[36].

A quick overview of practical considerations for different types of architecture, parallelism, and synchronization strategy can be accessed here [1].

3.4 Other Large-scale ASR datasets

Large-scale speech recognition has had varying definitions even over the past few years. In 2014, authors applied data and model parallelism for training RNN-based ASR models [23]. They used 5000 hours of labelled data to build a multi-GPU powered data-driven speech system which produced state-of-the-art results for that period of time. This solution was also quite robust to distortions and designed to work on both clear, conversational speech and also speech in noisy environments. The work also showed that with data parallelism on more than a few GPUs, scaling up the overall batch size linearly with the number of GPUs did not help with the model convergence rate.

More recently, research works from large enterprise companies have continued to focus on increasing the scale of training by multiple folds. Jinyu Li, et al. from Microsoft compare popular end to end speech recognition for large-scale ASR [32]. Authors consider an RNN transducer, an Attention-based encoder decoder, and transformer architectures and train them with 65,000 hours of training data both for streaming mode and non-streaming modes. In their experiments, to maintain a fair comparison, they have fixed the overall number of parameters to a single limit of around 87M. Their results show that transformer models obtained the best Word Error Rates (WER) outperforming the other models by around 1% WER. They also report that pretraining works well even for large-scale tasks, compared to random initialization.

Google has also worked on increasing their scale of data used for their experiments. In 2019, researches have published their work on building a domain invariant speech recognition model, using large amounts of data from multiple domains to train a single model that work across different applications, sampling rates, audio codecs and formats, and noise conditions [41]. They claim that their model can adapt to an entirely new domain with as little as one tenth of data compared to that required for a model trained from scratch. This helps to develop speech recognition models for practically any environment with less amount of data. Since then, recent Google research has used datasets that are a huge 300,000 hours in size [42]. The dataset consists of data from voice search, far field use cases, telephone speech, and audio from YouTube videos. This nature of multi-domain data is again useful for inter-domain speech recognition applications. They also compare the impact of increasing data diversity in improving performance of the models. The authors assess how best to use multi domain data when different proportions of data are available from different domains. They try training models

[1] Distributed training with TensorFlow, TensorFlow Core. https://www.tensorflow.org/guide/distributed_training

using equal probability from different domain and ignore the relative distribution of data among the domains. From their experiments, this usually tended to overfit the domains that had lesser data. Hence, the overall conclusion was to sample the data from all domain with the probability proportional to the total number of utterances in the domain. Overall, they report that using multi domain data and in large scales improved the output performance of the model significantly.

The previously discussed datasets are not available in public domain, and this has been one of the major issues for developing models for large-scale speech recognition jobs in academic backgrounds. Recent efforts [17, 52] have been to create such a publicly available dataset that can be used in academic research. The People's Speech dataset [17] is a free to download dataset with 31400 hours with more data continuously added. The data, collected by scraping the internet for appropriately licensed audio data with transcriptions. The data collected is diverse in nature, with varying sampling rates, background noise and different forms of speech. There are a few problems with this dataset, as the authors state that there is no defined measure if there are duplicates in the dataset because the dataset's creation is by crawling the internet, it is possible that some data appears twice in the dataset. This also means that creating test sets with no overlap with the training set is challenging.

Multilingual LibriSpeech (MLS) [52] dataset is the first attempt to create a large scale multilingual dataset. It has a total of 32,000 hours of English data and 4,500 hours of data from 7 other languages. The dataset is entirely audiobooks, with transcriptions downloaded from the internet and then aligned with the audio. This dataset can be used for both speech recognition and text to speech applications since the audio is quite clean with minimal to zero background noise.

Common Voice [3] is a crowd-sourced multilingual dataset that is mainly focused to propel automatic speech recognition tasks. This dataset is easily scalable for any language because community efforts drives both data collection and data validation. At the time of writing this report, the common voice dataset has 12,500 hours of labelled data [2]. For many of the languages in the common voice dataset, it is the only publicly available dataset source for speech recognition tasks. Users can record audio by reading out sentences or phrases that are shown to them through common voice's app or website. The text used in common voice comes mainly from Wikipedia articles. For data validation, a maximum of three volunteers listen to an audio clip and the clip is valid as soon as two votes are positive.

3.5 Summary

It is apparent from industry-based research that there are huge improvements and advantages to be gained from increasing the dataset size used for training the models. There is a lot of work done in this direction to scale up the amount of

[2]Common Voice. https://commonvoice.mozilla.org/en

publicly available speech data that can be used for academic research. *Hence, it is time for researchers in academic field to be well-prepared to scale up training by multiple folds in the near future. The aim of this thesis is to act as a guidebook for scaling up training in multiple steps of the training process.* The next chapters discuss the best methods to store data, loading of data, training using multi-GPUs, training using multiple processes and reporting performance.

Chapter 4

Implementation

4.1 Overview

In this chapter, we start off by discussing the dataset used and what makes the dataset unique in Section 4.2. Then, we look at how to make the data usable for an ASR application with steps like forced alignment and audio normalization in Section 4.3. Next, we go over the details for training the acoustic encoder decoder model in Section 4.4. Lastly, we explore the distributed methodologies used and discuss the practical choices taken when using those methods in Section 4.5

4.2 Business Speech (BizSpeech) Dataset

As mentioned in the Section 3.4, there are not many datasets which are available publicly for training speech recognition systems at a scale of tens of thousands of hours. This section introduces Business Speech dataset, which has been used for all the experiments in this thesis.

Business speech dataset consists of earnings calls, which are corporate disclosure and brokerage event information. It can be accessed through a paid licence here[1]. Refinitiv also provides transcriptions, call summaries and other metadata like date and PermID, which is the ID number for the firm. This ID can further be used to link to many more metadata information from other datasets like the officers and directors dataset[2] which provides information about the firm's financials, personnel involved etc. The audio and transcripts can be accessed via APIs and also via the Eikon software[3].

[1]Company Events Coverage, StreetEvents, Refinitiv. https://www.refinitiv.com/en/financial-data/company-data/company-events-coverage

[2]Officers and Directors data, Refinitiv https://www.refinitiv.com/en/financial-data/company-data/officers-and-director-search

[3]Eikon Financial Analysis & Trading Software, Refinitiv https://www.refinitiv.com

Figure 4.1 shows a snapshot of the software in use.

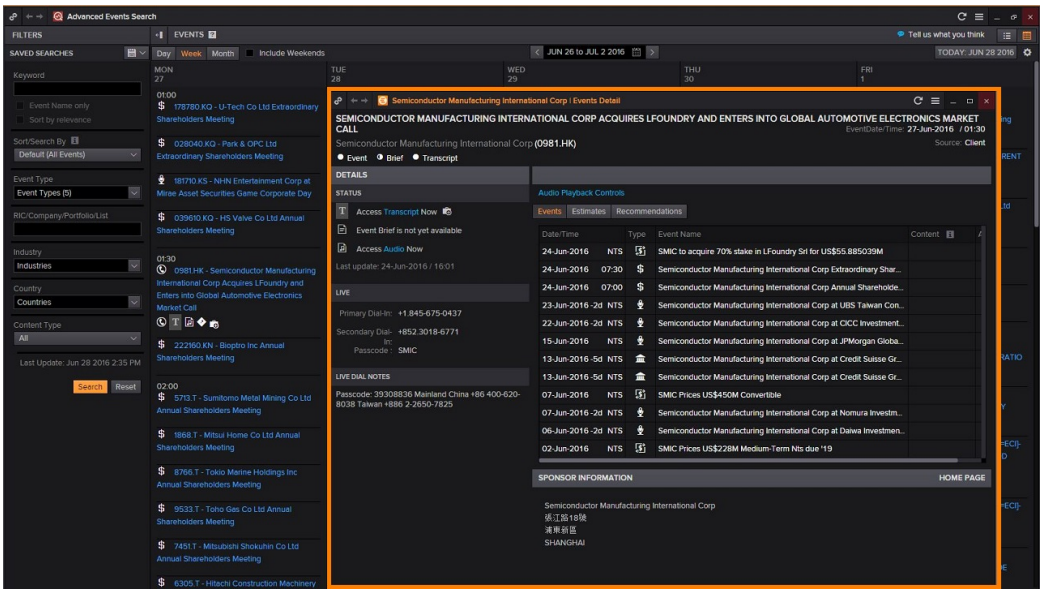

Figure 4.1: Conference call event information in Eikon.

The dataset which is used for the experiments consist of 24,793 conference calls held by 6,131 firms all over the world. All the calls are in English language. Earnings calls pertaining to fiscal year 2017 contribute to 68% of the dataset, 22% are from 2016 and 5% each is from 2015 and 2018. The conference calls are one-hour long conversations with the initial part of the call consists of a presentation about the firm's financials in the previous period and the rest of the call is a question and answer session with journalists and other analysts. Hence, the dataset has different types of speech, prepared one-sided speech is in the first half of the audio and the rest are more conversational in nature. The speech is mostly in the sphere of finance and business and consists of a lot of jargon from this domain. The dataset has diverse speech forms, from different English accents. An analysis of the companies' metadata show that the dataset consists of conference calls of companies from 76 countries. Out of the 24793 events, 4822 (around 20%) are from non-native English-speaking countries. Even among the native English-speaking countries, there is a wide distribution of events between the USA, UK, Australia, New Zealand, etc.

Since there are no previously available results on this exact dataset for the task of automatic speech recognition, a portion of it was processed using popular cloud speech to text services to analyse the complexity of the dataset. We use Azure's[4]

[4]Speech to Text, Audio to Text Translation, Microsoft Azure https://azure.microsoft.com/en-us/services/cognitive-services/speech-to-text/

speech to text service and Google's[5] speech to text services to analyse around 2000 randomly sampled utterances from the BizSpeech dataset (the same samples were used for both services). The Word Error Rate (WER) on the two services were 19.74% and 21.9% on Azure and Google, respectively. Furthermore, on analysing the word error rates based on the nationality of the companies, from Figure 4.2 we can observe that on Google, there is a drop of 32.7% in WER and in Azure, the drop is 36.5% from native to non-native utterances which shows that even the most generalized models can suffer with different speaking styles and accents.

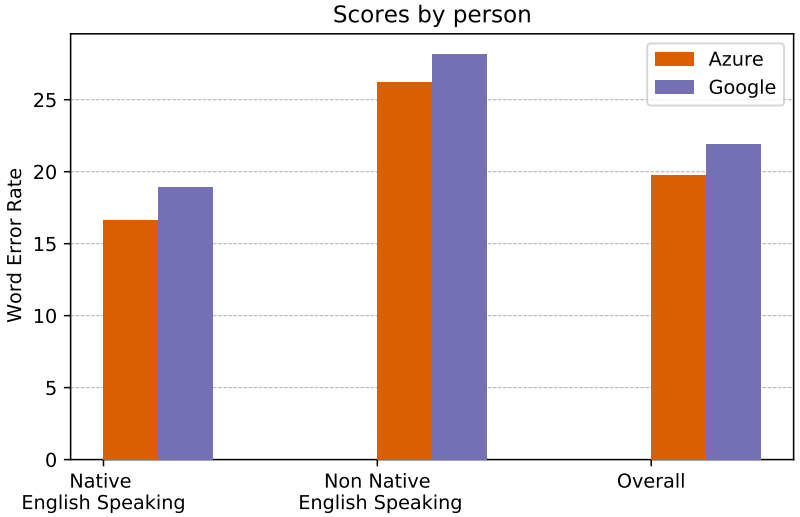

Figure 4.2: Word Error Rate on Google's and Azure's cloud services.

The main advantage of this dataset is that it is quite a large dataset and includes various speech styles and accents. The drawback of the dataset is that the speech is from a single domain and from a specific field, which might mean that the models trained on this dataset might not be useful for general speech recognition tasks.

The transcript in the dataset consists of text with punctuations and with standard capitalization applied. The dataset consists of non-standard text, like abbreviations, random characters to represent word fillers (like ah, eh, oh.) and many abbreviations and named entities. The phrase "operator instructions" is used in the transcript very frequently to denote that the conference operator is giving out instructions pertaining to the call, and the actual phrase is not present in the audio at all. These inconsistencies (examples shown in Table 4.1) make the dataset

[5]Speech-to-Text: Automatic Speech Recognition, Google Cloud https://cloud.google.com/speech-to-text/

harder to train and since the size of the dataset is large, it is hard to know where and what other inconsistencies are present in the dataset.

Punctuation	
Transcript:	`[..] with the short-term dynamics?`
Verbatim:	`[..] with the short term dynamics`
Model output:	`[..] with the short-term dynamics?`
Capitalization	
Transcript:	`Hi. This is Andy.`
Verbatim:	`hi this is andy`
Model output:	`Hi, this is Andy.`
Abbreviations & Named Entities	
Transcript:	`[..] a question from Andrew Porteous, HSBC.`
Verbatim:	`[..] a question from andrew porteous h s b c`
Model output:	`[..] a question from Andrew Porteous, HSBC.`
Random characters	
Transcript:	`[..] things are -- [theoretically], good`
Verbatim:	`[..] things are eh theoretically good`
Model output:	`[..] things -- theoretically, good`
Transcript inconsistencies	
Transcript:	`Operator Instructions`
Verbatim:	`welcome to the petit home q one results call`
Model output:	`Welcome to the Petit Home Q1 Results Call.`

Table 4.1: Examples of non-standard text in the dataset. For each example, we show the ground truth transcript, the verbatim transcript, which is exact audible speech from the audio file, and the ASR model output, which is the output of our trained end to end speech to text model.

SPGISpeech[46] dataset consists of 5,000 hours of transcribed data from corporate presentations. The dataset is similar in nature to the BizSpeech dataset, although, it is not as large as BizSpeech. Their transcriptions also consists of punctuation and capitalizations similar to BizSpeech. It is difficult to say if the dataset has any direct overlap with BizSpeech as the metadata about the conference calls are not available in detail with SPGISpeech. They also remove utterances with currency information (8%) on the grounds that utterances with currency terms are non-trivial to normalize. They publish the train and validation splits of the dataset and keep the test dataset private. Earnings-21 dataset [12] is another dataset which originate from conference calls. Their transcriptions also have punctuations and capitalizations in them. However, this dataset is small in comparison to SPGISpeech and BizSpeech, with only around 39.25 hours. The calls are all

from 2020, which means that there is no direct overlap with the BizSpeech dataset. Apart from ASR, this dataset can be used for Entity recognition tasks as it has the suitable annotations required for that task as well.

4.3 Data Preparation

After downloading the dataset, training ASR models still requires a few preprocessing steps. This section explains these preprocessing steps in more detail.

The audio recordings in the BizSpeech dataset, are a mix of MP3 and WAV files which have varying sampling rates between 8 kHz, 16 kHz and 32 kHz. We convert all the audio to WAV format with 16 kHz sampling rate to maintain uniformity across the whole dataset. Since the proportion of 8 kHz data in the whole dataset is quite low, we chose not to downscale everything to 8 kHz to normalise the audio. Upsampling data from 8 kHz to 16 kHz does not add any new information to the dataset and has no particular advantage other than to keep all the data uniform.

4.3.1 Forced Alignment

Even though BizSpeech has transcriptions for the speech, many of them lack timestamps for the utterances or are inaccurate when present. The audio files are around 1 hour long in length, and in practice a randomly initialized attention mechanism could not converge to attending to the relevant sections of the audio and the memory requirements to process an hour-long audio utterance would also prevent it from using it as is. Hence, we must have finer and accurate timestamps for the sentences to split the audio to one sentence length to use the data for training ASR models.

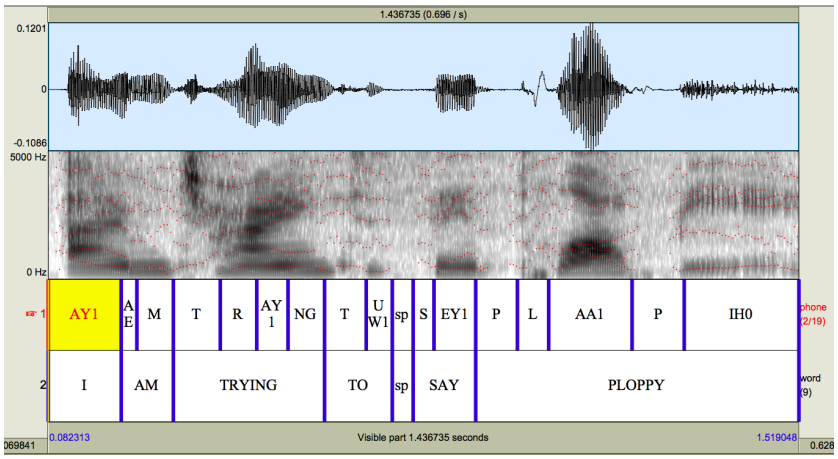

Figure 4.3: Result of forced alignment for an example input. [65]

We use forced alignment to generate accurate alignment of speech with the textual data. Forced alignment helps to align linguistic units (e.g., phoneme or words) with the corresponding audio file. It requires an audio file with transcriptions as input, and it outputs the text with word level time-aligned data. We use the P2FA for Python 3[6] tool, which is a HMM forced aligner based on P2FA [65] to generate alignments. The aligner works by using a HTK [64] based acoustic model along with a pronunciation dictionary. The tool generates the transcript's phoneme representation, and then the acoustic model is used to find the best matching time-alignment output. An example of this can be seen in Figure 4.3. As mentioned in their work, the errors for the successful alignments are around 50ms. A risk of using forced aligner is that it always tries to generate some sort of alignment. When any sorts of inconsistencies are present with the text and audio, for example, a missing word in the transcript, they can deceive the tool for the whole file's alignment and the tool fails. Due to this issue, *the usable data from the 24,793 audio hours for training ASR model is finally 19973 hours with close to 9 million utterances with a composition of 52,484 speakers.*

4.3.2 Data Loading Performance

Now that accurate time alignments are generated for the dataset, we split the large audio files into smaller files such that each audio split has one sentence in it. Each one of these splits is linguistically an utterance, and we label them as an individual utterance and store as a separate WAV file. We store the text for the utterance in a JSON file along with a few metadata fields, with a unique utterance ID as the filename for both the audio clip and the JSON file.

Each utterance length ranges from one second up to 60 seconds. Figure 4.4 shows the distribution of the utterance length for the whole dataset. We can see that most of the utterances are around the 6-second length. Since, the utterances are small there are about 9 million audio files and 9 million JSON files to be loaded for every training run. This becomes highly inefficient[7] and will lead to problems on opening too many files on the training node. The size of the dataset is also around 5 TB, which makes it hard to store on a single disk storage. Hence, we cannot depend on storing the data on the local SSD drive. All these problems lead to extremely high usage of disk storage, disk I/O usage, network costs (especially for multi node training), and high CPU usage to access and read a high number of files.

[6]Penn Phonetics Lab Forced Aligner Toolkit (P2FA) for Python3, https://github.com/jaekookang/p2fa_py3

[7]Small files, Aalto scientific computing https://scicomp.aalto.fi/triton/usage/ small-files/

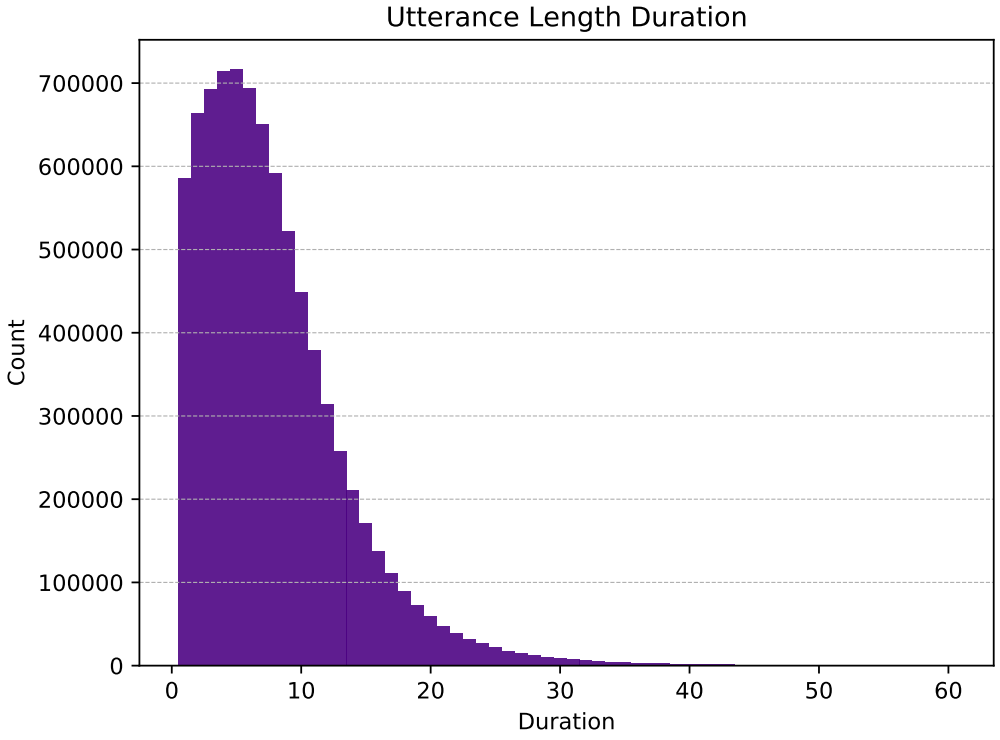

Figure 4.4: Utterance Duration Distribution for BizSpeech Dataset

Hence, we use WebDataset[8][2], a PyTorch Dataset (`IterableDataset`[9]) implementation which provides high efficiency access to data stored in TAR[10] archives. This uses only sequential/streaming data access, which brings substantial performance advantage. Figure 4.5 shows, on the left side, a file-based access to resources and on the right side, an illustration of sequential based access to resources. We can see from the illustration that sequential data access is much faster and requires much lesser communication. Typically, a ten-fold increase in performance can be seen during I/O operations[2] for a single node setup with sequential access. This enables large-scale training. Because, it supports basic TAR archives the other advantage is that it becomes easy to create, manage and distribute the data for deep

[8]webdataset/webdataset: A high-performance Python based I/O system for large (and small) deep learning problems, with strong support for PyTorch. https://github.com/webdataset/webdataset/

[9]torch.utils.data, PyTorch 1.9.0 documentation https://pytorch.org/docs/stable/data.html

[10]GNU tar 1.34: Basic Tar Format https://www.gnu.org/software/tar/manual/html_node/Standard.html

learning training. Since, TAR can be compressed using gzip[11] and this is supported directly by webdataset we can save storage space as well. Webdataset can also be set up to use sharding. Sharding helps achieve high throughput using parallel I/O for multiprocess enabled tasks like data loading, preprocessing, etc. This. We specify the shards as a list of files to webdataset, or they can be written using the brace notation. For example, `bizspeech-shard-{000000..002999}.tar` means there are 3000 shards. When used with a standard Torch `DataLoader`, this will perform parallel I/O and preprocessing.

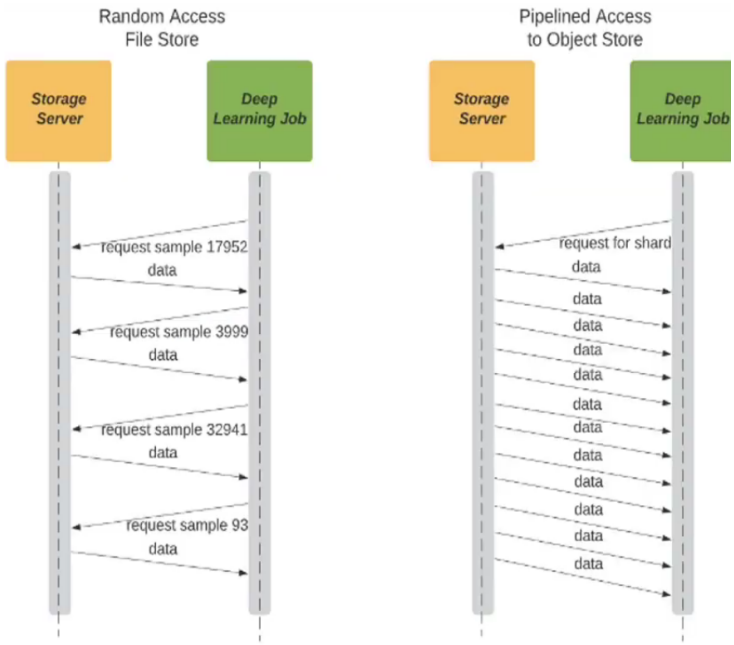

Figure 4.5: File-based Access vs Sequential Data Access [2]

We convert the BizSpeech dataset to compressed tar archives to enable usage of webdataset. Split the dataset into 3000 shards, with each shard containing about 700 MB of data after compression, totalling at around 2.1 TB storage footprint. This dataset is now ready to be used for training ASR models efficiently. Empirically, speed-up after using webdataset with a single process was around 7-8 fold during training a small subset of 80 hours of the dataset. It becomes very difficult practically to store and test the file-based mechanism to measure the throughput difference for very large-scale experiments. For all the large-scale experiments of order more than 200 hours, we use a configuration of 2-4 worker processes per GPU with sharding and randomization enabled.

[11] gzip, Wikipedia https://en.wikipedia.org/wiki/Gzip

4.4 Training an AED model

For training a speech to text AED model on BizSpeech dataset, we use SpeechBrain (introduced in Section 2.4). SpeechBrain provides recipes which are designed and optimized for particular datasets. Considering the scale of experiments, we use the LibriSpeech recipe[12] as the starting point. After data loading and preprocessing to parse the audio and transcript text, we generate features from the audio signal. We use logarithmic, mel-based, filter banks [61] with 0-8000 Hz frequency and the processing happens on-the-fly with data loading. Data loading can be parallelized easily by enabling multiprocessing to increase the number of available workers to load the data. From the recipe, we disable the external language model for our experiments because the focus is on scaling up the training of speech to text and not necessarily to make the model the most accurate one. The final model and configuration used is given in the Appendix A. We use the negative log likelihood loss function along with the CTC loss (described in Section 2.3.4.1) for the initial 15 epochs.

4.4.1 Dynamic Batching

A problem in automatic speech recognition is when batching up the utterances for training, the varying lengths of utterances means that the shorter utterances have to be padded up to the length of the longest utterance in the batch. Depending on the ordering of the utterances, the batches of data used for training could be high in sparsity due to padding. Because the ordering of training data is usually random, a fixed batch size can be inefficient when the utterances are short. To counter this, we could sort the speech data in length to make the GPU usage more efficient. However, this may affect the model convergence based on the type of ordering used. To solve these problems, SpeechBrain uses dynamic batch data loader where it reads the audio samples to the memory buffer. It then clusters the audio samples based on length and then generates batches of audio samples from similar length utterances, hence avoiding the inefficiencies in data loading due to padding issues.

4.4.2 *Epoch* with dynamic batching

The data is in random order, and since the data loading workers access the shards asynchronously, it is difficult to keep track of exact epochs (exact epoch here refers to each data sample used exactly once per epoch). Hence, we drop the ”each sample once per epoch” standard, and instead use sampling with replacement where we draw a sample completely randomly. Previous experiments have shown

[12] speechbrain/recipes/LibriSpeech/ASR/seq2seq https://github.com/speechbrain/speechbrain/tree/develop/recipes/LibriSpeech/ASR/seq2seq

that sampling by replacement is as good as or in some cases better than each sample once per epoch strategy [44, 54]. Tracking the training progress generally involves observing the loss values and accuracy metrics of a validation step after every epoch of training, and since with the newer approach, traditional "epochs" are inconsequential, we track the number of updates to the model. We run the validation step after every 5000 updates to the model. Henceforth, *epoch* refers to 5000 updates to the model and this is used to track training progress using validation loss and validation WER.

4.5 Distributed training

We use two different methods of Multi-GPU training, Distributed Data-Parallel (DDP) is a synchronous, decentralized and as the name suggests, a data-parallel method. Hogwild is an asynchronous, data parallel method.

4.5.1 Synchronous Training

We use PyTorch's distributed data parallel method to train our model [33]. This method enables training in distributed systems by aggregating gradients before the optimizer step is invoked. This ensures that all model replica parameters are updated using the same gradients, and thus the replicas stay synchronized across all the iterations. Hence, this can be equated to accumulating gradient across multiple batches of data on a single GPU. PyTorch uses `AllReduce` operation which is supported by primitive inter process communication libraries like NCCL[13], MPI[14], etc. The `AllReduce` operation expects a tensor of the same size from all of its participant process and applies a given arithmetic operation, and sends back the output tensor to all the processes involved. It is through this operation that DDP becomes a synchronous method because `AllReduce` waits until all processes complete for a particular iteration, and each individual process waits for the output from `AllReduce` before it moves to the next iteration. More specifically, for implementing DDP for our models, SpeechBrain wraps the models with a `DistributedDataParallel` module from PyTorch. The data loader explained in Section 4.3 scales without any modifications required for a distributed training use-case. We use different processes (can be more than one process per replica as well) to load data for the different model replicas.

[13]NVIDIA Collective Communications Library (NCCL), NVIDIA Developer https://developer.nvidia.com/nccl

[14]Open MPI: Open-Source High-Performance Computing https://www.open-mpi.org/

4.5.2 Asynchronous Training

We implement Hogwild [45], discussed in Section 3.3.2.2. We discuss the parallel processing steps in more details here. We assume a shared parameter model across p processors. The parameters are denoted by x and each process accesses x and can update x because x is stored in shared memory. To update the weights with an update a, there is no locking system enforced, so any process can perform the following operation on the shared memory.

$$x_v \leftarrow x_v + a$$

Once the training starts, each processor follows the pseudocode provided in Algorithm 1. It loads the current state of the model, x_e. Let $G_e(x)$ denote a gradient of the function f_e, as in standard SGD, for a data point e sampled randomly from the dataset, E. γ is the step size and controls the magnitude of the gradient update. The parameters are updated directly without blocking the other processes.

Algorithm 1: Hogwild algorithm for each process.

while do
  Sample e uniformly at random from E
  Read current state x_e and evaluate $G_e(x)$
  $x_v \leftarrow x_e - \gamma G_e(x)$
end

Once updated, the loop continues with its next iteration by loading new data and loading newer parameters from the shared memory. Since there is no blocking of the model parameters, sometimes it could so happen that an update of the parameters is overwritten without being used by any processors at all. Results from [45] however show that the training can benefit from the non-blocking nature of the training procedure. This is mainly because of the robustness of SGD to the random ordering of updates to the parameters. SGD is invariant to the data order and hence to the order in which the updates to the model are received.

Chapter 5

Experiments & Evaluations

5.1 Overview

In this chapter, we start with exploring how we evaluate the experiments and what metrics we use to measure them in Section 5.3. Next, in Section 5.2 we discuss the experimental setup and the system used for the experiments. To better understand the impact of data in ASR experiments, we run multiple training configurations using different amounts of data in Section 5.5. Then, in Section 5.6 we analyse the impact of distributed training and evaluate the impact of using multi GPU, multiprocess techniques on training time and WER. Next, we try to match hyperparameters of distributed training exactly with normal training to check the direct time comparisons of benchmark time. In Section 5.7, we compare the runs for 20,000 hours dataset with different methods of training. Next, we analyse the effect of accent of the WER in Section 5.8. Finally, we evaluate our model with other publicly available datasets and report the WERs on those test sets.

5.2 Experimental Setup

For all of our experiments, we use Aalto University's Triton[1], a high-performance computing cluster. Triton has an Nvidia DGX-1[2] machine which contain 8 Tesla V100 GPUs and are optimized for deep learning applications. Each GPU has a 32 GB memory with CUDA compute 7.0 capability. These GPUs are used for all the deep learning training jobs, including the multi-GPU ones. This research work also required large amounts of storage that can be accessed from multi processes and from multiple nodes at the same time, we use the Lustre[3] filesystem which

[1]Triton cluster, Aalto scientific computing https://scicomp.aalto.fi/triton/
[2]Nvidia DGX, Wikipedia https://en.wikipedia.org/wiki/Nvidia_DGX
[3]Lustre https://www.lustre.org/

is a scalable filesystem, which provides large storage capacity and high sequential throughput for cluster-based applications. The lustre storage is connected via an Infiniband[4] connection, which features high throughput and low latency.

5.3 Evaluation Criteria

We monitor the training jobs using TensorBoard. The main metrics that we track are training loss value, validation loss and validation WER. The input transcript follows the typical conventions for capitalization and punctuation, as discussed in Section 4.2. In the initial stages of experiments, we did not take steps to remove punctuation marks and capitalization from the input text, and this led to the acoustic model being trained end to end along with the capitalization and punctuation marks in them. Hence, for the later experiments, we stuck with this setup because we noticed that the model was able to predict the capitalization and punctuation marks in many cases. During post-processing, we apply text normalization and report word error rates based on the test split of the dataset, with and without punctuations or capitalization. Therefore, for all experiments we have a word error rate with punctuations considered (WER-P) and without punctuations (WER) in them. We also report the Character Error Rate (CER) with punctuations for the test set. Table 5.1 show the train, validation, and test data splits details.

	Utterances	Hours
`Train`	8809282	20000
`Validation`	186165	413
`Test`	185623	412

Table 5.1: Dataset Train Validation Test Statistics

5.4 Experiments overview

We use `SentencePiece` Tokenizer with vocabulary size of 5k with *bpe* as the token type. The detokenized output from the acoustic model consists of upper case characters, lower case characters, digits, punctuation marks, percent sign, currency notations, space character, a token for filler words ("- -"), hence covering all the characters present in the dataset transcriptions. The model used is an attention-based encoder decoder model. For the encoder, we use a CRDNN model. The convolution part of the CRDNN model has 2 blocks. Each block has 2 convolution layers with `LayerNorm` applied and `LeakyReLU` activation function. These two

[4]InfiniBand, Wikipedia. https://en.wikipedia.org/wiki/InfiniBand

convolution layers are then followed by `MaxPool` and `Dropout` layers. The two major blocks are chained together, and then the output is passed to a bidirectional-LSTM cell with 512 neurons. The output of the recurrent part of the neural network is passed to the MLP portion of the network. This has two blocks of Linear Layer with `BatchNorm` and `LeakyReLU` activation function with `Dropout`. We then use the output of the encoder is fed to the decoder part of the network. The decoder is a location aware attentional RNN with a GRU cell with the attention dimension set to 512. Appendix A shows the complete model architecture and complete hyperparameter configuration in sown in B. We use Adam optimizer with learning rate set to 0.0001 without any scheduler. We apply CTC loss for the initial 15 epochs and then disabled it for further training. The final model is the checkpoint with the best validation word error rate. The target batch size used was 180 seconds, with a maximum batch size of 240 seconds. The batch size is represented in seconds because of the dynamic batching method explained in Section 4.4.1. We do not use external language models to evaluate the final metrics.

5.5 Dataset size-based experiments

We performed experiments to compare the models between the different scales of data used for training jobs. We run training jobs using 80, 200, 2000, 8000 and 20000 hours. For all the data scales, we spawn two processes for data loading and preprocessing. The different dataset splits that were used are shown in Table 5.2. All the experiments are run using the same hyperparameters.

Note that for evaluation, we use the same test and validation sets across all the scales to make it convenient to compare the results. Since we create the test and validation splits with the largest dataset (20000 hours) in mind, they are larger than the training sets for 80 and 200 hour datasets. Predictably, the word error rates drop with the increase in the scale of training data, and we see a similar trend with the character error rates. The difference between the word error rates with and without text normalization is at an almost constant 3.5% points. The CER-P is also an important metric for our dataset because of the presence of punctuation and capitalization in the transcript. We also report training time to benchmark. This is measured by having a benchmark WER of 19.74%, which is the Azure's mean WER on random utterances from the dataset, discussed in Section 4.2. So, *time to benchmark here is defined as the training wall clock time for the model to reach the validation word error rate of 19.74%.* We see that the larger dataset reached the benchmark WER before the smaller 8000 hour dataset. We hypothesize that the more diverse dataset helped to generalize better, and this led to quicker and better convergence of the model. This further showcases the importance of large-scale training tasks, even when training time is an important factor taken into consideration.

Hours	Utterances	WER	WER-P	CER-P	Time to Benchmark
80	38246	46.73%	50.38%	29.63%	42+
200	94840	43.84%	47.84%	29.08%	64+
2000	964824	33.76%	38.52%	27.8%	140+
8000	3480514	20.74%	24.31%	12.04%	76
20000	8809282	14.01%	17.12%	7.67%	48

Table 5.2: Dataset Split for training at different scales. The Hours column shows the number of hours of training data used for the experiments. The Utterances column shows the number of training utterances for each data split. WER is word error rate without punctuations, WER-P is the word error rate considering the punctuations and capitalizations, CER-P is the character error rate with the punctuations and capitalizations. Time to benchmark is the wall clock time taken for the training to reach the benchmark word error rate. "+" indicates that the hours reported is the total training time and not the time to benchmark because the run never reached the benchmark accuracy.

5.6 Distributed Training

All the experiments discussed above are run using a single GPU and a single process. This section now delves into the distributed training experiments.

5.6.1 Synchronous training

We use the Distributed Data-Parallel (DDP) approach, which is a Data parallel, synchronous training method which uses the `allreduce` operation to keep the weights synchronized. In these experiments, we spawn 4 processes, each with access to a discrete GPU. The same model is copied to all the GPUs. Each process computes the forward pass and the loss and gradients to backpropogate, and for each GPU, we also spawn two processes for data loading and preprocessing on the fly. For using multiple GPUs, the batch size becomes a vital hyperparameter. If we retain the same batch size as that was used for the single GPU, the overall effective batch size is the number of GPUs used times the single batch size. Table 5.3 shows the word error rates for these experiments. For the DDP experiments, we use 4 GPUs, with the effective batch size hence becomes 4 times the individual batch size.

We compare the distributed data parallel results with the single GPU results. We observe improvement of 15%-38% in WER. This can be attributed to the fact that larger batch sizes work better for the hyperparameter configuration and model architecture setup because everything apart from that is kept constant. Hence, the use of DDP is perfect for these experiments since it enables usage

Train Data Size (in hrs)	8000			20000		
	WER	WER-P	CER-P	WER	WER-P	CER-P
Single GPU	20.74%	24.31%	12.04%	14.01%	17.12%	7.67%
4-GPU DDP	17.38%	20.65%	10.16%	10.87%	13.69%	5.73%

Table 5.3: WER, WER-P and CER-P comparison for single GPU and multi GPU runs.

of larger batch sizes. From these experiments, it is apparent that using DDP is beneficial, especially in cases where larger batch sizes lead to better convergence of the models.

	WER	WER-P	CER-P	Time to Benchmark
1-GPU	20.74%	24.31%	12.04%	**59**
1-GPU SGA	16.74%	19.98%	7.60%	**80**
4-GPU DDP	17.38%	20.65%	7.67%	**42**

Table 5.4: WER, WER-P and CER-P comparison for single GPU, single GPU with sequential gradient accumulation and multi GPU runs for the 8000 training hours dataset. The table also shows the wall clock time to benchmark with the different setups.

Hence, to make comparisons on the time to benchmark of the models, we try to level the batch sizes for the two setups. One way would be to reduce the batch size by a factor of the number of GPUs used for the DDP setup. However, this means that the efficiency of the training takes a hit. Therefore, we choose a different method to achieve consistency between the setups. Since the gradients in the multiple GPU setup are updated 4 batches at a time by computing them in parallel, we apply the backward pass after accumulating the gradients for 4 batches, processed sequentially using a single GPU. This effectively means that the batch size is increased by a factor of 4. We call this method Sequential Gradient Accumulation (SGA). In this method, we compute the forward pass, the loss, and the gradients for four batches sequentially and then run the backward passes together after processing the four batches. This makes the comparison more fair and easier to make. We conduct this experiment for the 8000 training hour dataset. We expect similar results with the larger dataset as well.

We compare the word error rate for the SGA experiment with the other results in the Table 5.4. We see that the word error rates differ by around 0.5% points between the SGA and the DDP methods, and this can be attributed to minute variations like initial seeds and other random occurrences. We also report the time to benchmark in the same table. We see that the single GPU run reached the benchmark WER at around the 60 hours mark, which is 1.5 times the time to benchmark of the 4-GPU DDP, but the single GPU model doesn't converge overall

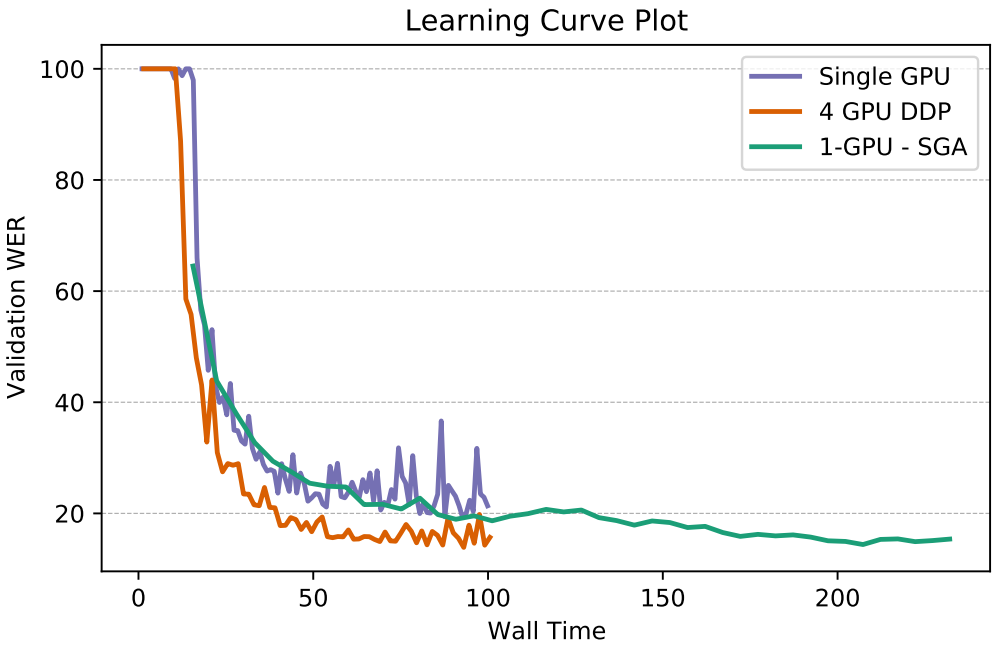

Figure 5.1: Learning curves during training for the different setups, with Validation WER on the y-axis and wall clock time in hours on the x-axis.

as much as the DDP run. Comparing the 1-GPU SGA and the 4-GPU DDP, the time to benchmark is half. The single GPU SGA takes 80 hours to reach the benchmark WER. Figure 5.1 shows the learning curve for the three runs, with the validation word error rate on the y-axis and wall clock time in hours on the x-axis. Note that we stop the training when the training loss was stagnant over at least 5 epochs. Again we see that the DDP plot reaches the benchmark word error rate the quickest and converges sharply in the initial hours.

5.6.2 Asynchronous training

Hogwild [45] is used to evaluate the usage of asynchronous training. In this experiment, we store the weights on one single GPU, and we spawned 3 different processes to process the forward pass over different batches of data and calculate the loss and gradients to backpropogate. Then each process continues to update the weights on the GPU. Next, all individual processes again read the updated weights and continues to iterate over the next batches of data. We use 2 processes for each main process for data loading and preprocessing. Overall, the system works without any interlocking mechanism to ensure that the updates to

the model weights are actually read by the other processes before being overwritten. Again, the batch size is a crucial hyperparameter because all the 3 processes use the same GPU, the same batch size as single process cannot be used as we will face issues managing the GPU memory. If we reduce the batch size for the single process experiment, the device efficiency reduces. Table 5.5 shows the word error rates for these experiments. For the Hogwild experiments, we use 3 processes, and since each process acts on the same GPU, the batch size should be divided by the number of processes to keep the memory consumption about the same level, and thus the effective batch size is 1/3rd of the single GPU batch size.

Train Data Size (in hrs)	**8000**			**20000**		
	WER	WER-P	CER-P	WER	WER-P	CER-P
1-Process	20.74%	24.31%	12.04%	17.38%	20.65%	10.16%
Hogwild	22.4%	25.7%	12.70%	24.09%	27.56%	13.81%

Table 5.5: WER, WER-P and CER-P comparison for single process and 3 process distributed Hogwild runs for two data scales

We compare the Hogwild results with the single GPU results. We observe a deterioration of 8%-38% in WER with using Hogwild. This can be attributed to the fact that smaller batch sizes work does not work well for the hyperparameter configuration and model architecture setup because all the other configurations is the same. Hogwild also doesn't improve by adding more data. We see that the accuracy metrics are better for the 8000 hours training set. Similar to the synchronous training experiments, comparing these results which have different effective batch sizes is not reasonable.

	WER	WER-P	CER-P	Time to Benchmark
1-Process	20.74%	24.31%	12.04%	**35**
1-Process $\frac{1}{3}$ BS	24.16%	27.75%	14.91%	**54**
3-Process Hogwild	22.4%	25.7%	12.70%	**46**

Table 5.6: WER, WER-P and CER-P comparison for single GPU, single process with one third batch size and multi GPU runs for the 8000 training hours dataset. The table also shows the wall clock time to benchmark with the different setups.

Hence, to make the comparisons on the time to benchmark of the models, we use sequential updates to simulate multiple process settings on a single process system. We reduce the batch size of the single process experiment to one third the size of the regular batch size, and this equates it to the Hogwild experiment closely. This makes the comparison more fair and easier to make. We conduct this experiment for the 8000 training hour dataset to get the results faster. We expect

similar results with the larger dataset as well. Since Hogwild never reaches WER of 19.74%, we use a word error rate of 30% as the benchmark to compare the time to benchmark. Table 5.6 shows the word error rate for the single process with reduced batch size, compared to the other results, along with wall clock time to benchmark (at 30% WER) in hours. We see that Hogwild does slightly better in word error rates and character error rates than the sequential update with reduced batch size. Comparing the time to benchmark, we see that the single process run converges quickest, but along the experiments which uses the same effective batch size, Hogwild converges around 8 hrs quicker than the sequential update simulation of Hogwild.

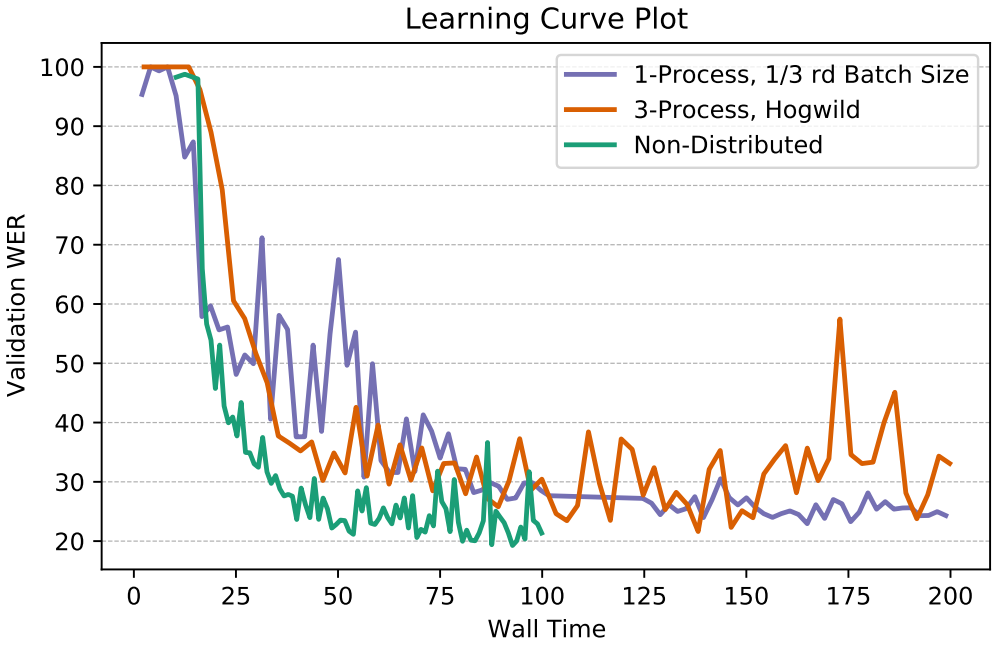

Figure 5.2: Learning curves during training for the different setups, with Validation WER on the y-axis and wall clock time in hours on the x-axis.

Figure 5.2 shows the learning curve for the two runs, with the validation word error rate on the y-axis and wall clock time in hours on the x-axis. We stop the training, when the training loss is stagnant over at least 5 epochs. The learning curves are quite erratic, and we observe a lot of variation in validation word error rates even when the training loss vales are constant.

5.7 Comparing the results for 20k dataset

Our experiments show the different methods to accelerate all the different parts of the pipeline for training ASR models for data scalability. We now report the word error rate, word error rate with punctuations and capitalizations considered and the character error rate among the three different methodologies. The first method is the centralized training with a single GPU with multiprocessing used only for data loading and preprocessing. The second methodology is the Distributed Data Parallel method, which is the synchronous training using 4 GPUs, with one process on each GPU for training. The third method is the asynchronous training with Hogwild, with 3 processes using a single GPU to share weights and update them in parallel. These results are reported in Table 5.7. but to compare the training methods for large-scale datasets and to showcase the best workflow for such experiments, we now compare the different metrics for the largest training dataset size, i.e. 20,000 hours.

	WER	WER-P	CER-P	Training Wall-time (hrs)
1-GPU	14.01%	17.12%	7.67%	**59**
4-GPU DDP	10.87%	13.69%	5.73%	**32**
1-GPU Hogwild	24.09%	27.56%	13.81%	**100+**

Table 5.7: WER, WER-P and CER-P comparison for single GPU, single GPU, multi GPU DDP, 1-GPU Hogwild runs for the 20,000 training hours dataset. The table also shows the wall clock time to benchmark with the different setups. "+" indicates the total training time and not the time to benchmark because the run never reached the benchmark accuracy.

The synchronous DDP training job with WER of 10.87% and CER-P of 5.73% is the best result obtained overall. Synchronous training methods work well when the different GPUs are similar in their processing power and memory storage. This avoids stragglers and training synchronization can happen smoothly. This can also be seen in our experiments that the time to benchmark is reduced by half with the use of DDP on multiple GPUs.

Some caution has to be taken before reading these results, because the hyperparameters were optimized only once at the beginning of the experiments, and it is possible that for the different methods, varying the hyperparameters could lead to better results. The main aim of this thesis was not to get the best WER for the dataset but to evaluate the best practices and methods for data scalability, and hence this step was not given much prominence during the experiments.

5.8 Effect of accent on the results

In Section 4.2, we had discussed the challenges in the dataset concerning the utterances which are spoken by people from different parts of the world, including non-native speakers of English, which is particularly interesting and useful for building a diverse model. We compare the word error rates for our best model with Google's & Azure's results and also analyse the drop in accuracy between native and non-native utterances in Figure 5.3

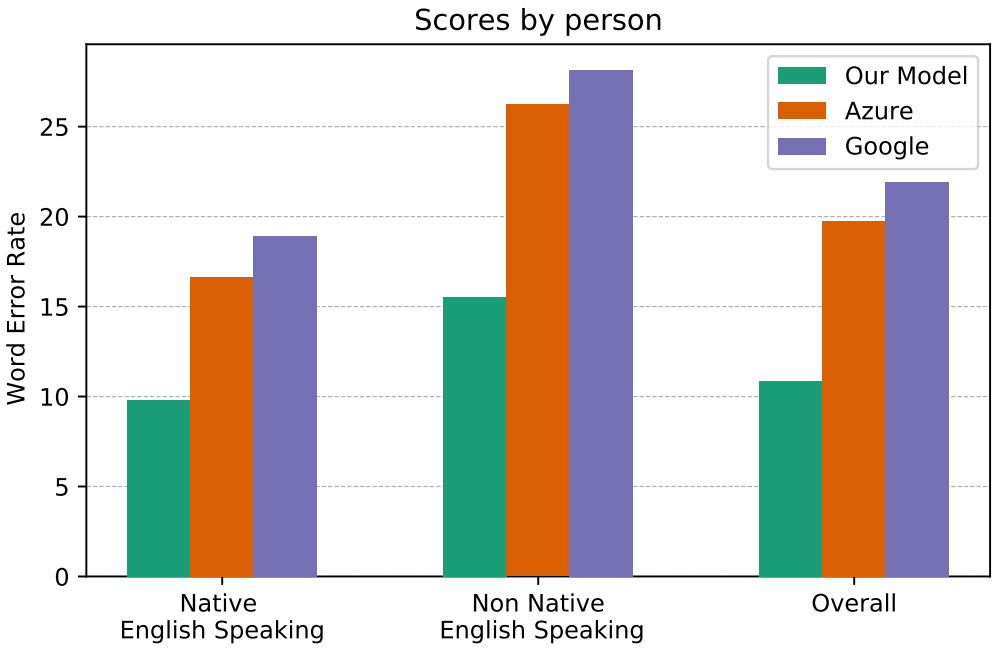

Figure 5.3: Word error rates comparison for native and non-native speakers of English between Google, Azure and our models.

The word error rates for the native and non-native speakers for our model are calculated on different utterances from what we used with Google's and Azure's services. This is due to overlap between the utterances used for cloud services to estimate WER and the training set of our model. There is a consistent degradation of performance pertaining to non-native English speakers.

5.9 Evaluation with other datasets

Here we report the results with our models for other datasets and compare them to the metrics with our models.

5.9.1 SPGISpeech

SPGISpeech [46], as discussed in Section 4.2 is quite similar to BizSpeech dataset, and we evaluate our best model on their dataset. The authors of SPGISpeech have reported their results on the `test` split of the dataset, but have made only the `val` and `train` splits public. Hence, we evaluate our model on the `val` split of the model. Table 5.8 shows the WER and WER-P for the `val` split on our model along with the results from the Conformer [20] model evaluated on the `test` split from SPGISpeech experiments.

Results	WER	WER-P
Our Model	8.66%	15.26%
SPGISpeech [46]	2.3%	5.7%

Table 5.8: WER and WER-P comparison for val split of SPGISpeech using our best model and test split with the Conformer model from SPGISpeech experiments.

The normalized WER on SPGISpeech of 8.66% is less than the best word error rate obtained on BizSpeech, which indicates that the data in the SPGISpeech is much easier, because of the removal of data with currencies in them etc. The normalized WER from SPGISpeech is reported after not only removing the casing of the text and punctuations, but also reduce the vocabulary while post-processing the text output to report the WER results. It is also important to note that the results are not on the same data split, and this will have an impact on the WERs. Since, the test split that was used by the authors is not available, it is not possible to resolve this difference.

5.9.2 LibriSpeech

Since the dataset size is big, we wanted to evaluate the model's generalization capabilities on a dataset which is not related to BizSpeech. We employ similar strategy as the researchers of the People's Speech Dataset [17]. We evaluate our best model on the LibriSpeech[47] test datasets without any transfer learning, so the LibriSpeech data is completely unseen data for the model. Table 5.9 shows the test word error rates for LibriSpeech datasets and compare it with the People's Speech Dataset results. People's speech dataset consists of 30000 hours of diverse speech ranging from audiobooks, news, movies, commentaries, court hearings, etc.

This makes the People's dataset quite close to the BizSpeech dataset, because of its size and diverse nature. Since LibriSpeech data does not contain any punctuations and capitalization, we normalize the hypothesis text from our model before calculating the WER metrics.

Results	**test-clean**		**test-other**	
	WER	CER	WER	CER
Our Model	22.46%	10.62%	37.93%	20.41%
People's Speech [17]	32.17%	18.86%	57.56%	41.2%

Table 5.9: WER and CER comparison for test-clean and test-other datasets with the People's Speech model

Although, the results with our model are worse than state-of-the-art models on LibriSpeech, we manage to outscore the people's speech models by 30% on the test-clean dataset and by 35% on the test-other dataset which shows the model can generalize pretty well.

Chapter 6

Discussion

We discuss the research problems laid out in Section 1.1, and then we go over the next steps in the research related to both the dataset and also related to the model architectures and experiments in Section 6.2.

6.1 Revisiting the problem statements

1. **How to enable large-scale training tasks from an engineering point of view? What are the best methods to store data, load data for such training jobs?**

   We use TAR archives to store and access data directly from them using WebDataset. This enables sequential access to training data and makes the data loading efficient and compatible with multiple workers in parallel.

2. **What is the effect of training data scale on the quality of the acoustic model used?**

   Increasing the scale of training data always led to huge boost in WER and other evaluation metrics. It raises a question of whether there is one point of performance beyond which adding more data does not help with improving the model performance.

   (a) **Does increasing the scale of training data affect the convergence time of the acoustic model?**

   Increasing the scale of training data actually led to *reduction* in the training time to reach the benchmark WER. This showed the effect of having diverse and large dataset to obtain better model performance.

3. **How effective are the multi-GPUs and multiple processing jobs to accelerate the training time and reach convergence quicker?**

Distributed training methods helped not only to reach benchmark WER quicker, but also to improve the overall performance, of the model. The effective batch size used in these experiments played a huge role in the quality of the models.

(a) **Compare synchronous and asynchronous training methods and analyse which of them are suitable for our system setup?**

Synchronous data parallel training method proved to be well suited for our experiment setup and worked efficiently to reduce the training time and also provided better WER in than non-distributed training. In contrast, asynchronous training did not fare too well in our experiments, although this could change with further hyperparameter optimization.

(b) **What is the speed-up achieved by using these methods for training an acoustic end to end model?**

For synchronous training, a speed-up of close to 2x was observed when training with 4 GPUs in parallel.

6.2 Future Work

6.2.1 Dataset related work

The dataset used is large and this has enabled all the experiments that we have conducted, but the dataset is not without its drawbacks and future work can be done to address some of these inconsistencies. There are phrases like "Operator Instructions" which are discussed in Section 4.2 which are wrong transcriptions of what is being said in the audio, and this only came to our attention in the later stages of our experiments. Although some preprocessing steps were taken, the dataset needs more thorough cleaning to remove utterances like the above example from the dataset, which can then lead to better models.

The dataset can also be used for a wide variety of other applications like named entity recognition when properly tagged and in other domains to infer business related results based on the speech in the earnings calls. Also, due to the varied nature of the speakers, it can also be filtered and used to train models which are designed to work with a diverse set of speakers.

6.2.2 Further ASR Experiments

Changing the model architecture can be tried and experiments involving scaling both model and data can be analysed and the correlation between the two scaling methods can be studied in detail. This could help establish a practical formula which can help researchers decide what minimum amount of data will be

required to train models with certain number of parameters. Experiments can also focus on deeper and wider layers in the AED model. Other architectures like transformers[60], conformers[20] can be explored for large-scale ASR experiments. This could be easy to extend using the same pipeline, but by replacing the model architecture with various other models.

With more time, we would have liked to tune hyperparameters for each type of training and analyse the best word error rates from the different techniques. Currently, the results indicate that methods that work with large batch sizes have fared well and models seem to struggle when batch size is reduced. It will be interesting to see if this holds even after varying the other hyperparameters like the learning rate, adding a learning rate scheduler, different optimizers etc. In all our experiments, the acoustic models are trained end-to-end with punctuations, capitalizations and other special characters. This is one of the biggest unknown in the experiments, about how the model would vary if it was trained with the normalized text instead.

6.2.3 Evaluation Methods

For evaluating the models, we choose the best validation word error rate checkpoint as the final model to be used with the test set. However, in many cases we observed a sharp decline between the validation word error rates and the test word error rates even when both the sets are unseen and sampled randomly. This can be avoided by techniques like averaging checkpoints over multiple epochs, which have competitive error rates. This method should be used in the future experiments.

Chapter 7

Conclusions

Deep learning has set the trend in the previous decade for automating tasks like automatic speech recognition, and data scalability is one of the paths to improve the models. Large-scale training experiments are challenging to carry out due to lack of resources and infrastructure availability, and it becomes crucial to make use of the available assets efficiently.

We introduce Business speech dataset with around 9 Million utterances and 26,000 hours in size. The dataset is quite unique considering the diverse nature of speakers in it, and it also has two forms, conversational and prepared speech in it. Data scalability promises high performance, but also comes with its challenges. As the scale of the data increases, the chances of data having inconsistencies also go up. This is hard to keep track of, especially manually.

We presented solutions which enable training models using data scales in the order of few Terabytes. Sequential access to the dataset through TAR archives is paramount even to be able to practically run experiments above 2000 hours of data. In our case, we use WebDataset to achieve this, and it also integrates nicely with PyTorch. It also supports accessing from multiple nodes, processes and to use with multiple GPUs which help us with the distributed training setups.

We use an Attention based Encoder-Decoder architecture as the default for all the experiments and compare three different types of training strategies.

In the non distributed training jobs, the best WER of the models dropped consistently as we increase the scale of the data used for training, with the best WER of 14.01% on the 20,000-hour dataset. We also observed the importance of using large-scale data as the model trained with the 20,000-hour dataset reached the benchmark WER in 48 hours compared to 76 hours with the 8000-hour dataset. This means that using larger datasets does not necessarily mean longer training times. It can be argued that it is better to train with large datasets, even when the time required to complete the training is a crucial factor for the experiments.

Synchronous training works best in our experiments and provide the best word error rates. We observed a speed-up of 2x when using 4-GPUs in data parallel mode. The best WER is 10.87% with the 20000 hours dataset and using 4-GPU

DDP. Hence, we can conclude that synchronous training methods are effective, especially when the hardware setup (Similar GPUs, CPUs in a multinode environment) is homogeneous in nature, so that the straggler problem becomes irrelevant. Asynchronous training methods did not fare well in our experiments, but it could improve with hyperparameter optimization focussed for that strategy of training. Even though there was a slight improvement in convergence time, the performance metrics were considerably worse than the other methodologies. The WER learning curve for the asynchronous training also does not inspire confidence for the method, as it was very erratic even when the loss value was constant.

In this particular thesis, we provide a full workflow for speech recognition with large-scale data to speed up training. We discuss methods for data storing, data loading and then to get maximum efficiency of resources available by enabling usage of parallelization techniques involving multiple GPUs and processes. The work can be accessed on GitHub[1].

[1]aalto-speech/BizSpeech_SpeechBrain: Building an ASR system recipe for BizSpeech data using SpeechBrain. https://github.com/aalto-speech/BizSpeech_SpeechBrain

Bibliography

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

[2] Aizman, A., Maltby, G., and Breuel, T. High Performance I/O for Large Scale Deep Learning. *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019* (12 2019), 5965–5967.

[3] Ardila, R., Branson, M., Davis, K., Henretty, M., Kohler, M., Meyer, J., Morais, R., Saunders, L., Tyers, F. M., and Weber, G. Common Voice: A Massively-Multilingual Speech Corpus. *Proceedings of the 12th Language Resources and Evaluation Conference* (2020).

[4] Chan, W., Jaitly, N., Le, Q., and Vinyals, O. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings 2016-May* (5 2016), 4960–4964.

[5] Chilimbi, T., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project Adam: Building an Efficient and Scalable Deep Learning Training System. *Big Learning Workshop* (2014), 571–582.

[6] Chiu, C.-C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, E., Jaitly, N., Li, B., Chorowski, J., and Bacchiani, M. State-of-the-art Speech Recognition With Sequence-to-Sequence Models. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings 2018-April* (12 2017), 4774–4778.

[7] Cipar, J., Ho, Q., Kim, J. K., Lee, S., Ganger, G. R., Gibson, G., Keeton, K., and Xing, E. Solving the Straggler Problem with Bounded Staleness. *14th Workshop on Hot Topics in Operating Systems (HotOS {XIV})* (2013).

[8] Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Urgen Schmidhuber, J. Flexible, High Performance Convolutional Neural Networks for Image Classification. *Twenty-second international joint conference on artificial intelligence* (2011).

[9] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. *Proceedings of the Eleventh European Conference on Computer Systems* (2016).

[10] Dahl, G. E., Yu, D., Deng, L., and Acero, A. Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition. *IEEE Transactions on Audio, Speech, and Language Processing 20*, 1 (2012).

[11] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, A., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large Scale Distributed Deep Networks. *Advances in neural information processing systems 25* (2012), 1223–1231.

[12] Del Rio, M., Delworth, N., Westerman, R., Huang, M., Bhandari, N., Palakapilly, J., McNamara, Q., Dong, J., Zelasko, P., and Jette, M. Earnings-21: A Practical Benchmark for ASR in the Wild. *arXiv:2104.11348* (4 2021).

[13] Deng, L., Li, J., Huang, J.-T., Yao, K., Yu, D., Seide, F., Seltzer, M. L., Zweig, G., He, X., Williams, J., Gong, Y., and Acero, A. Recent Advances in Deep Learning for Speech Research at Microsoft. *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), 8604–8608.

[14] Deshmukh, A. Comparison of Hidden Markov Model and Recurrent Neural Network in Automatic Speech Recognition. *pdfs.semanticscholar.org 5*, 8 (2020).

[15] Deyringer, V., Fraser, A., Schmid, H., and Okita, T. Parallelization of Neural Network Training for NLP with Hogwild! *Prague Bull. Math. Linguistics 109* (2017), 29–38.

[16] Enarvi, S. Modeling Conversational Finnish for Automatic Speech Recognition. *dipl Sähkötekniikan korkeakoulu ELEC* (2018), 117 + app. 73.

[17] Galvez, D., Diamos, G., Manuel, J., Torres, C., Ai, F., Achorn, K., Gopi, A., Kanter, D., Lam, M., Mazumder, M., and Reddi, V. J. The People's Speech: A Large-Scale Diverse English Speech Recognition Dataset for Commercial Usage. *35th Conference on Neural Information Processing Systems (NeurIPS 2021* (2021).

[18] Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to Forget: Continual Prediction with LSTM. *Neural Computation 12*, 10 (10 2000), 2451–2471.

[19] Graves, A., Mohamed, A. R., and Hinton, G. Speech recognition with deep recurrent neural networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (10 2013), 6645–6649.

[20] Gulati, A., Qin, J., Chiu, C.-C., Parmar, N., Zhang, Y., Yu, J., Han, W., Wang, S., Zhang, Z., Wu, Y., and Pang, R. Conformer: Convolution-augmented Transformer for Speech Recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2020-October* (5 2020), 5036–5040.

[21] Hagner, J. Recurrent Neural Networks for End-to-End Speech Recognition A comparison of gated units in an acoustic model. *cs.umu.se* (2017).

[22] Hammer, B., and Gersmann, K. A Note on the Universal Approximation Capability of Support Vector Machines. *Neural Processing Letters 2003 17:1 17*, 1 (2 2003), 43–53.

[23] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and Ng, A. Y. Deep Speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (12 2014).

[24] Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., Mostofa Ali Patwary, M., Yang, Y., and Zhou, Y. Deep Learning Scaling is Predictable, Empirically. *arXiv preprint arXiv:1712.00409* (2017).

[25] Hochreiter, S., and Schmidhuber, J. Long Short-Term Memory. *Neural Computation 9*, 8 (11 1997), 1735–1780.

[26] Jia, Z., Zaharia, M., and Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. *arXiv preprint arXiv:1807.05358* (7 2018).

[27] Kannan, A., Datta, A., Sainath, T. N., Weinstein, E., Ramabhadran, B., Wu, Y., Bapna, A., Chen, Z., and Lee, S. Large-Scale Multilingual Speech Recognition with a Streaming End-to-End Model. *Proceedings*

of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2019-September (9 2019), 2130–2134.

[28] Koliousis, A., Weidlich, M., Mai, L., Costa, P., and Pietzuch, P. CROSSBOW: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *arxiv.org* (2019).

[29] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (4 2014).

[30] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems 25* (2012), 1097–1105.

[31] Langer, M., He, Z., Rahayu, W., and Xue, Y. Distributed Training of Deep Learning Models: A Taxonomic Perspective. *IEEE Transactions on Parallel and Distributed Systems 31*, 12 (7 2020), 2802–2818.

[32] Li, J., Wu, Y., Gaur, Y., Wang, C., Zhao, R., and Liu, S. On the Comparison of Popular End-to-End Models for Large Scale Speech Recognition. *arXiv preprint arXiv:2005.14327* (2020).

[33] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., Chintala, S., and No-ordhuis, P. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *PyTorch Distributed: Experi-ences on Accelerating Data Parallel Training. PVLDB 13*, 12 (2020), 2150–8097.

[34] Mai, L., Li, G., Wagenländer, M., Fertakis, K., Brabete, A.-O., and Pietzuch, P. KungFu: Making Training in Distributed Machine Learning Adaptive. *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), 937–954.

[35] Mansikkaniemi, A. Acoustic model and language model adaptation for a mobile dictation service. *dipl Sähkötekniikan korkeakoulu ELEC* (2010).

[36] Mayer, R., and Jacobsen, H.-A. Scalable Deep Learning on Distributed Infrastructures. *ACM Computing Surveys (CSUR) 53*, 1 (2 2020).

[37] Mayer, R., Mayer, C., and Laich, L. The tensor flow partitioning and scheduling problem: It's the critical path! *DIDL 2017 - Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning, Part of Middleware 2017* (12 2017), 1–6.

[38] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device Placement Optimization with Reinforcement Learning. *International Conference on Machine Learning* (7 2017), 2430–2439.

[39] Morgan, N. Deep and Wide: Multiple Layers in Automatic Speech Recognition. *IEEE Transactions on Audio, Speech, and Language Processing 20*, 1 (2012), 7.

[40] Mukhedkar, A. S., and Alex, J. S. R. Robust feature extraction methods for speech recognition in noisy environments. *1st International Conference on Networks and Soft Computing, ICNSC 2014 - Proceedings* (2014), 295–299.

[41] Narayanan, A., Misra, A., Sim, K. C., Pundak, G., Tripathi, A., Elfeky, M., Haghani, P., Strohman, T., and Bacchiani, M. Toward Domain-Invariant Speech Recognition via Large Scale Training. *2018 IEEE Spoken Language Technology Workshop, SLT 2018 - Proceedings* (2019), 441–447.

[42] Narayanan, A., Prabhavalkar, R., Chiu, C.-C., Rybach, D., Sainath, T. N., and Google, T. S. Recognizing long-form speech using streaming end-to-end models. *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)* (2019).

[43] Ney, H. J., and Ortmanns, S. Dynamic programming search for continuous speech recognition. *IEEE Signal Processing Magazine 16*, 5 (1999), 64–83.

[44] Nielsen, M. A. Neural Networks and Deep Learning. *http://neuralnetworksanddeeplearning.com/* (2015).

[45] Niu, F., Recht, B., Ré, C., and Wright, S. J. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *arXiv preprint arXiv:1106.5730* (2011).

[46] O'neill, P. K., Lavrukhin, V., Majumdar, S., Noroozi, V., Zhang, Y., Kuchaiev, O., Balam, J., Dovzhenko, Y., Freyberg, K., Shulman, M. D., Ginsburg, B., Watanabe, S., and Kucsko, G. SPGISpeech: 5,000 hours of transcribed financial audio for fully formatted end-to-end speech recognition. *arXiv:preprint arXiv:2104.02014v2* (2021).

[47] Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. Librispeech: An ASR corpus based on public domain audio books. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings 2015-August* (8 2015), 5206–5210.

[48] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32* (2019).

[49] Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. *Proceedings of the 13th EuroSys Conference, EuroSys 2018 2018-January* (4 2018), 14.

[50] Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlíček, P., Qian, Y., Schwarz, P., SilovskÂ´y, J. S., Stemmer, G., and VeselÂ´y, K. V. The Kaldi Speech Recognition Toolkit. *IEEE 2011 workshop on automatic speech recognition and understanding* (2011).

[51] Prabhavalkar, R., Rao, K., Sainath, T. N., Li, B., Johnson, L., and Jaitly, N. A Comparison of Sequence-to-Sequence Models for Speech Recognition. *Interspeech* (2017), 939–943.

[52] Pratap, V., Xu, Q., Sriram, A., Synnaeve, G., and Collobert, R. MLS: A Large-Scale Multilingual Dataset for Speech Research. *arXiv preprint arXiv:2012.03411* (2020).

[53] Ravanelli, M., Parcollet, T., Plantinga, P., Rouhe, A., Cornell, S., Lugosch, L., Subakan, C., Dawalatabad, N., Heba, A., Zhong, J., Chou, J.-C., Yeh, S.-L., Fu, S.-W., Liao, C.-F., Rastorgueva, E., Grondin, F., Aris, W., Na, H., Gao, Y., De Mori, R., and Bengio, Y. SpeechBrain: A General-Purpose Speech Toolkit. *arXiv preprint arXiv:2106.04624* (2021).

[54] Recht, B., and Ré, C. Beneath the valley of the noncommutative arithmetic-geometric mean inequality: conjectures, case-studies, and consequences. *arXiv preprint arXiv:1202.4184* (2012).

[55] Robbins, H., and Monro, S. A Stochastic Approximation Method. *The Annals of Mathematical Statistics 22*, 3 (1951), 400–407.

[56] Sayed, E., Ahmed, A., Saeed, R., and Saeed, R. A. A Survey of Big Data Cloud Computing Security Geoinformatics Technology and Application in Sudan View project Wireless Mesh Network (P.0000091) View project A Survey of Big Data Cloud Computing Security. *Article in International Journal of Computer Science International Journal of Computer Science and Software Engineering (IJCSSE) 3*, 1 (2014).

[57] Schuster, M., and Paliwal, K. K. Bidirectional Recurrent Neural Networks. *IEEE Transactions on Signal Processing 45*, 11 (1997).

[58] Tanaka, T., Masumura, R., Ihori, M., Takashima, A., Orihashi, S., and Makishima, N. End-to-End Rich Transcription-Style Automatic Speech Recognition with Semi-Supervised Learning. *arxiv.org* (7 2021).

[59] Valiant, L. G. A Bridging Model for Parallel Computation. *Communications of the ACM 33*, 8 (1 1990), 103–111.

[60] Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Å., and Polosukhin, I. Attention Is All You Need. *Advances in neural information processing systems* (2017), 5998–6008.

[61] Vetterli, M., and Herley, C. Wavelets and filter banks: Theory and design. *infoscience.epfl.ch* (1992).

[62] Watanabe, S., Hori, T., Karita, S., Hayashi, T., Nishitoba, J., Unno, Y., Soplin, N. E. Y., Heymann, J., Wiesner, M., Chen, N., Renduchintala, A., and Ochiai, T. ESPnet: End-to-End Speech Processing Toolkit. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2018-September* (3 2018), 2207–2211.

[63] Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., Yang, F., and Zhou, L. Gandiva: Introspective Cluster Scheduling for Deep Learning. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (2018), 595–610.

[64] Young, S., Evermann, G., Kershaw, D., Moore, G., Odell, J., Ollason, D., Povey, D., Valtchev, V., and Woodland, P. The HTK Book. *Cambridge university engineering department* (2002).

[65] Yuan, J., and Liberman, M. Speaker identification on the SCOTUS corpus. *ling.upenn.edu* (2008).

[66] Zhang, H., Xu, S., Dai, W., Liang, X., Hu, Z., Wei, J., Xie, P., Zheng, Z., Ho, Q., and Xing, E. P. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. *2017 USENIX Annual Technical Conference* (2017).

[67] Zhang, Y., Chan, W., and Jaitly, N. Very deep convolutional networks for end-to-end speech recognition. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* (6 2017), 4845–4849.

Appendix A

Model architecture

Encoder Architecture

```
CRDNN(
  (CNN): Sequential(
    (block_0): CNN_Block(
      (conv_1): Conv2d(
        (conv): Conv2d(1, 128, kernel_size=(3, 3),
                       stride=(1, 1))
      )
      (norm_1): LayerNorm(
        (norm): LayerNorm((40, 128), eps=1e-05,
                          elementwise_affine=True)
      )
      (act_1): LeakyReLU(negative_slope=0.01)
      (conv_2): Conv2d(
        (conv): Conv2d(128, 128, kernel_size=(3, 3),
                       stride=(1, 1))
      )
      (norm_2): LayerNorm(
        (norm): LayerNorm((40, 128), eps=1e-05,
                       elementwise_affine=True)
      )
      (act_2): LeakyReLU(negative_slope=0.01)
      (pooling): Pooling1d(
        (pool_layer): MaxPool2d(kernel_size=(1, 2),
                          stride=(1, 2), padding=(0, 0),
                          dilation=(1, 1), ceil_mode=False)
      )
      (drop): Dropout2d(
```

```
      (drop): Dropout2d(p=0.15, inplace=False)
    )
  )
  (block_1): CNN_Block(
    (conv_1): Conv2d(
      (conv): Conv2d(128, 256, kernel_size=(3, 3),
                     stride=(1, 1))
    )
    (norm_1): LayerNorm(
      (norm): LayerNorm((20, 256),
                        eps=1e-05, elementwise_affine=True)
    )
    (act_1): LeakyReLU(negative_slope=0.01)
    (conv_2): Conv2d(
      (conv): Conv2d(256, 256, kernel_size=(3, 3),
                     stride=(1, 1))
    )
    (norm_2): LayerNorm(
      (norm): LayerNorm((20, 256), eps=1e-05,
                     elementwise_affine=True)
    )
    (act_2): LeakyReLU(negative_slope=0.01)
    (pooling): Pooling1d(
      (pool_layer): MaxPool2d(kernel_size=(1, 2),
                        stride=(1, 2), padding=(0, 0),
                        dilation=(1, 1), ceil_mode=False)
    )
    (drop): Dropout2d(
      (drop): Dropout2d(p=0.15, inplace=False)
    )
  )
)
(time_pooling): Pooling1d(
  (pool_layer): MaxPool2d(kernel_size=(1, 4),
                  stride=(1, 4), padding=(0, 0),
                  dilation=(1, 1), ceil_mode=False)
)
(RNN): LSTM(
  (rnn): LSTM(2560, 512, num_layers=4,
          batch_first=True, dropout=0.15, bidirectional=True)
)
(DNN): Sequential(
  (block_0): DNN_Block(
```

```
      (linear): Linear(
        (w): Linear(in_features=1024,
                    out_features=256, bias=True)
      )
      (norm): BatchNorm1d(
        (norm): BatchNorm1d(256, eps=1e-05,
                    momentum=0.1, affine=True,
                    track_running_stats=True)
      )
      (act): LeakyReLU(negative_slope=0.01)
      (dropout): Dropout(p=0.15, inplace=False)
    )
    (block_1): DNN_Block(
      (linear): Linear(
        (w): Linear(in_features=256, out_features=256,
                    bias=True)
      )
      (norm): BatchNorm1d(
        (norm): BatchNorm1d(256, eps=1e-05,
                        momentum=0.1, affine=True,
                        track_running_stats=True)
      )
      (act): LeakyReLU(negative_slope=0.01)
      (dropout): Dropout(p=0.15, inplace=False)
    )
  )
)
```

Decoder Architecture

```
AttentionalRNNDecoder(
  (proj): Linear(in_features=1024, out_features=512, bias=True)
  (attn): LocationAwareAttention(
    (mlp_enc): Linear(in_features=256,
                    out_features=512, bias=True)
    (mlp_dec): Linear(in_features=512,
                    out_features=512, bias=True)
    (mlp_attn): Linear(in_features=512,
                    out_features=1, bias=False)
    (conv_loc): Conv1d(1, 10, kernel_size=(201,),
                    stride=(1,), padding=(100,),
                    bias=False)
```

```
    (mlp_loc): Linear(in_features=10,
                      out_features=512, bias=True)
    (mlp_out): Linear(in_features=256,
                      out_features=512, bias=True)
    (softmax): Softmax(dim=-1)
  )
  (drop): Dropout(p=0.15, inplace=False)
  (rnn): GRUCell(
    (rnn_cells): ModuleList(
      (0): GRUCell(640, 512)
    )
    (dropout_layers): ModuleList()
  )
)
```

Appendix B

Configuration Files

dataset.yaml

```
seed: 98585
__set_seed: !apply:torch.manual_seed [!ref <seed>]
dataset_seed: 26000

output_folder: !ref runs/<seed>
save_folder: !ref <output_folder>/save
local_dataset_folder: !ref /m/triton/scratch/
    biz/bizspeech/ASR_Datasets/<dataset_seed>/datasets

# Path where data manifest files will be stored
train_annotation: !ref <local_dataset_folder>/train.json
valid_annotation: !ref <local_dataset_folder>/val.json
test_annotation: !ref <local_dataset_folder>/train.json

# Webdataset Parameters
use_wds: True
number_of_shards: 3000
#shard_maxcount: 10000
shardfiles_pattern: !ref <local_dataset_folder>/
    bizspeech_shard-%06d.tar.gz
use_compression: True
train_shards: (0, 2994)
val_shards: (2994, 2997)
test_shards: (2997, 3000)

# Audio Resampling and convert to single channel
sample_rate: 16000
```

```
preprocess_audio: !new:speechbrain.dataio.
            preprocess.AudioNormalizer
    sample_rate: !ref <sample_rate>
    mix: avg-to-mono

# Set up folders for reading from and writing to
data_folder: /scratch/biz/bizspeech/MEDIA
hours_reqd: 26000
nonnative: 0.3
qna: 0.6
sorting: random # ascending
non_CEO_utt: True
strict_included: False
utterance_duration_limit: 1000 # in seconds
audio_filetype: wav
trainValTest: (1,0)
```

tokenizer.yaml

```
dataset: !include:dataset.yaml

# Tokenizer parameters
token_type: bpe  # ["unigram", "bpe", "char"]
token_output: 5000  # index(blank/eos/bos/unk) = 0
character_coverage: 1.0
annotation_read: txt # field to read

# Tokenizer train object
tokenizer_train: !name:speechbrain.
    tokenizers.SentencePiece.SentencePiece
   model_dir: !ref <dataset[save_folder]>
   vocab_size: !ref <token_output>
   annotation_train: !ref <dataset[train_annotation]>
   annotation_read: !ref <annotation_read>
   model_type: !ref <token_type> # ["unigram", "bpe", "char"]
   character_coverage: !ref <character_coverage>
   annotation_list_to_check:
     - !ref <dataset[train_annotation]>
     - !ref <dataset[valid_annotation]>
   annotation_format: json
```

train.yaml

```
dataset: !include:dataset_200.yaml
tokenizer_params: !include:tokenizer.yaml

train_WER_required: False
wer_file: !ref <dataset[output_folder]>/wer.txt
wer_file_p: !ref <dataset[output_folder]>/wer_p.txt
train_log: !ref <dataset[output_folder]>/train_log.txt
tensorboard_dir: !ref runs/tensorboard_log/<dataset[seed]>

tokenizer: !new:sentencepiece.SentencePieceProcessor

pretrainer: !new:speechbrain.
    utils.parameter_transfer.Pretrainer
   collect_in: !ref <dataset[save_folder]>
   loadables:
      tokenizer: !ref <tokenizer>
      #model: !ref <model>
   paths:
      tokenizer: !ref <dataset[save_folder]>/
        <tokenizer_params[token_output]>_
        <tokenizer_params[token_type]>.model

gradient_accumulation: False
subbatches_count_for_grad_acc: 4
require_native_wer: False
wer_native_file: !ref <dataset[output_folder]>
    /wer_native.txt
wer_nonnative_file: !ref <dataset[output_folder]>
    /wer_nonnative.txt
libri:
    test_on_librispeech: False
    data_folder: librispeech
    test_csv:
        - !ref <libri[data_folder]>/test-clean.csv
        - !ref <libri[data_folder]>/test-other.csv
    tokenizer: !ref <tokenizer>
    bos_index: 0
    eos_index: 0
    test_dataloader_opts:
        batch_size: 8
        num_workers: 1
```

```
train_logger: !new:speechbrain.
    utils.train_logger.FileTrainLogger
    save_file: !ref <train_log>

log_to_tensorboard: True
tensorboard_logger: !new:speechbrain.
    utils.train_logger.TensorboardLogger
    save_dir: !ref <tensorboard_dir>

# Training parameters
number_of_epochs: 50
number_of_ctc_epochs: 15
batch_size: 8
lr: 0.0001
ctc_weight: 0.5
ckpt_interval_minutes: 15
label_smoothing: 0.1

# Dataloader options
train_dataloader_opts:
    batch_size: null
    num_workers: 8
valid_dataloader_opts:
    batch_size: null
    num_workers: 1
test_dataloader_opts:
    batch_size: null
    num_workers: 1

use_dynamic_batch_size: True
looped_nominal_epoch: 5000
dynamic_batch_kwargs:
    len_key: "sig"
    min_sample_len: 15999
    # 1s * 16000
    max_sample_len: 960000
     # 60s * 16000 (Sample rate)
    sampler_kwargs:
        target_batch_numel: 2880000
        # 180s * 16000 (Sample rate)
        max_batch_numel: 3840000
        # 240s * 16000 (Sample rate)
```

```
# Feature parameters
n_fft: 400
n_mels: 40

# Model parameters
activation: !name:torch.nn.LeakyReLU
dropout: 0.15
cnn_blocks: 2
cnn_channels: (128, 256)
inter_layer_pooling_size: (2, 2)
cnn_kernelsize: (3, 3)
time_pooling_size: 4
rnn_class: !name:speechbrain.nnet.RNN.LSTM
rnn_layers: 4
rnn_neurons: 512
rnn_bidirectional: True
dnn_blocks: 2
dnn_neurons: 256
emb_size: 128
dec_neurons: 512
output_neurons: 5000
blank_index: 0
bos_index: 0
eos_index: 0
unk_index: 0

# Decoding parameters
min_decode_ratio: 0.0
max_decode_ratio: 1.0
valid_beam_size: 80
#test_beam_size: 80
eos_threshold: 1.5
using_max_attn_shift: True
max_attn_shift: 240
lm_weight: 0.50
ctc_weight_decode: 0.0
coverage_penalty: 1.5
temperature: 1.25

epoch_counter: !new:speechbrain.
    utils.epoch_loop.EpochCounter
```

```
    limit: !ref <number_of_epochs>



# Feature extraction
compute_features: !new:speechbrain.
    lobes.features.Fbank
    sample_rate: !ref <dataset[sample_rate]>
    n_fft: !ref <n_fft>
    n_mels: !ref <n_mels>

# Feature normalization (mean and std)
normalize: !new:speechbrain.
    processing.features.InputNormalization

encoder: !new:speechbrain.
        lobes.models.CRDNN.CRDNN
    input_shape: [null, null, !ref <n_mels>]
    activation: !ref <activation>
    dropout: !ref <dropout>
    cnn_blocks: !ref <cnn_blocks>
    cnn_channels: !ref <cnn_channels>
    cnn_kernelsize: !ref <cnn_kernelsize>
    inter_layer_pooling_size:
        !ref <inter_layer_pooling_size>
    time_pooling: True
    using_2d_pooling: False
    time_pooling_size: !ref <time_pooling_size>
    rnn_class: !ref <rnn_class>
    rnn_layers: !ref <rnn_layers>
    rnn_neurons: !ref <rnn_neurons>
    rnn_bidirectional: !ref <rnn_bidirectional>
    rnn_re_init: True
    dnn_blocks: !ref <dnn_blocks>
    dnn_neurons: !ref <dnn_neurons>
    use_rnnp: False

embedding: !new:speechbrain.
    nnet.embedding.Embedding
    num_embeddings: !ref <output_neurons>
    embedding_dim: !ref <emb_size>

# Attention-based RNN decoder.
```

```
decoder: !new:speechbrain.
    nnet.RNN.AttentionalRNNDecoder
    enc_dim: !ref <dnn_neurons>
    input_size: !ref <emb_size>
    rnn_type: gru
    attn_type: location
    hidden_size: !ref <dec_neurons>
    attn_dim: 512
    num_layers: 1
    scaling: 1.0
    channels: 10
    kernel_size: 100
    re_init: True
    dropout: !ref <dropout>

# Linear transformation on the top of the encoder.
ctc_lin: !new:speechbrain.nnet.linear.Linear
    input_size: !ref <dnn_neurons>
    n_neurons: !ref <output_neurons>

# Linear transformation on the top of the decoder.
seq_lin: !new:speechbrain.nnet.linear.Linear
    input_size: !ref <dec_neurons>
    n_neurons: !ref <output_neurons>

# Final softmax (for log posteriors computation).
log_softmax: !new:speechbrain.nnet.activations.Softmax
    apply_log: True

# Cost definition for the CTC part.
ctc_cost: !name:speechbrain.nnet.losses.ctc_loss
    blank_index: !ref <blank_index>


modules:
    encoder: !ref <encoder>
    embedding: !ref <embedding>
    decoder: !ref <decoder>
    ctc_lin: !ref <ctc_lin>
    seq_lin: !ref <seq_lin>
    normalize: !ref <normalize>

model: !new:torch.nn.ModuleList
```

```
    - - !ref <encoder>
      - !ref <embedding>
      - !ref <decoder>
      - !ref <ctc_lin>
      - !ref <seq_lin>

valid_search: !new:speechbrain.
        decoders.S2SRNNGreedySearcher
    embedding: !ref <embedding>
    decoder: !ref <decoder>
    linear: !ref <seq_lin>
    bos_index: !ref <bos_index>
    eos_index: !ref <eos_index>
    min_decode_ratio: !ref <min_decode_ratio>
    max_decode_ratio: !ref <max_decode_ratio>

test_search: !new:speechbrain.
        decoders.S2SRNNGreedySearcher
    embedding: !ref <embedding>
    decoder: !ref <decoder>
    linear: !ref <seq_lin>
    bos_index: !ref <bos_index>
    eos_index: !ref <eos_index>
    min_decode_ratio: !ref <min_decode_ratio>
    max_decode_ratio: !ref <max_decode_ratio>

opt_class: !name:torch.optim.Adam
    lr: !ref <lr>

error_rate_computer: !name:speechbrain.
    utils.metric_stats.ErrorRateStats

cer_computer: !name:speechbrain.
    utils.metric_stats.ErrorRateStats
    split_tokens: True

checkpointer: !new:speechbrain.
    utils.checkpoints.Checkpointer
    checkpoints_dir: !ref <dataset[save_folder]>
    recoverables:
        model: !ref <model>
        scheduler: !ref <lr_annealing>
        normalizer: !ref <normalize>
```