TRANSCRIBING REAL-VALUED SEQUENCES WITH DEEP NEURAL NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Awni Hannun
March 2018

Abstract

Speech recognition and arrhythmia detection from electrocardiograms are examples of problems which can be formulated as transcribing real-valued sequences. These problems have traditionally been solved with frameworks like the Hidden Markov Model. To generalize well, these models rely on carefully hand engineered building blocks. More general, end-to-end neural networks capable of learning from much larger datasets can achieve lower error rates. However, getting these models to work well in practice has other challenges.

In this work, we present end-to-end models for transcribing real-valued sequences and discuss several applications of these models. The first is detecting abnormal heart activity in electrocardiograms. The second is large vocabulary continuous speech recognition. Finally, we investigate the tasks of keyword spotting and voice activity detection. In all cases we show how to scale high capacity models to unprecedentedly large datasets. With these techniques we can achieve performance comparable to that of human experts for both arrhythmia detection and speech recognition and state-of-the-art error rates in speech recognition for multiple languages.

Acknowledgements

So many people have contributed to the completion of my PhD. It's safe to say that this thesis would not have been possible were it not for the great collaborators I've had the good fortune of working with as well as my family and friends who supported me every day.

First, I'd like to thank my family. My entire family and especially my parents, Lina and Yusuf, have always unconditionally supported me in my career choices and inspired me in my work. I am also very grateful to my wife Kathy who first inspired me to come back to school while I was working at Google and has been with me every day since.

I would also like to thank my advisor Andrew Ng and co-advisor Dan Jurafsky. Dan's constant positive encouragement, enthusiasm and mentorship over the past six years have made the research not only possible but enjoyable.

I owe so much of my success in this work now and in the future to Andrew directly or indirectly given that he has trained so many of my close mentors and collaborators. Andrew has been a guide to me on all possible fronts including technically and strategically in research and in all of the softer skills which are critical to doing impactful work.

My first foray into machine learning research was during my time at Google, where I had the pleasure to be mentored by Quoc Le. Quoc introduced me to the just burgeoning field of deep learning and Andrew Ng's lab at Stanford. For this I am very grateful. I am also grateful to Andrew Maas for taking me under his wing during my first two years at Stanford. Andrew M. taught me a lot about machine learning and speech recognition. During that period I also had the good fortune of getting to know and work with many people in Andrew Ng's lab including Chris Lengerich, Peng Qi, Richard Socher, Brody Huval, Ziang Xie, Anshul Samar, Tao Wang, Sameep Tandon and Adam Coates.

I then spent close to two years as a Research Scientist at Baidu's newly formed Silicon Valley AI lab under the leadership of Adam Coates and Andrew Ng. This was a life changing experience for me and I couldn't be more grateful to have had Adam Coates as a close mentor and friend during that time and since.

I'm grateful to the entire Deep Speech team. As I now know, it's a rare thing indeed to work with such a highly productive and enjoyable group. The Deep Speech collaborators include Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates and Andrew Ng. Thank you all for being a part of such an exciting time and teaching me so much then and since. I would also like to thank the Deep Speech 2 team at Baidu and all of my collaborators there. There are too many to list invidually, but a few others that I worked closely with during that time include Dario Amodei, Jesse Engel, Eric Battenberg, Sherjil Ozair, Chong Wang, Tony Han, Jim Fan and Zhenyao Zhu.

I would also like to thank Chris Lengerich for being my collaborator in the start-up world and on many research projects over the years.

After returning to Stanford from Baidu Research, I ventured into machine learning applications for healthcare. During this time I had the good fortune to work with Pranav Rajpurkar, Geoffrey Tison, Mintu Turakhia and several employees at iRhythm Technologies including Masoumeh Haghpanahi, Codie Bourn and Justin Cambra. I am also grateful to my office mates Anand Avati and Ziang Xie as well as many other lab members including Sudnya Diamos, Dillon Laird, Jeremy Irvin, James Liu and Will Hang.

I'd like to thank Swati Dube for kicking-off and organizing many fruitful partnerships between our lab at Stanford and other organizations. Also thanks to the lab administrative staff and PhD student services staff for all their help over the years.

Thanks to my reading committee James Zou, Anshul Kundaje, Dan and Andrew for their feedback and advice as well as my thesis defense chair Nigam Shah.

Thanks to everyone who made this work possible through collaboration, mentorship, support, encouragement and of course friendship and entertainment.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Countless problems can be formulated as transcribing a real-valued sequence. Some commonly studied examples include automatic speech recognition and handwriting recognition. Reading electrocardiograms is another example from the medical domain. In this work we develop a general framework for building high accuracy models to transcribe real-valued sequences. Our framework requires two critical ingredients: a differentiable model based on deep neural networks and a sufficiently large annotated dataset. These ingredients can also be viewed as our primary generalization knobs. The more resources we spend on finding an accurate model and improving the size and quality of our annotated dataset, the better the overall performance of the model. In many cases, we must bring to bear techniques from high performance computing (HPC) to efficiently train these models.

In this work we exclusively rely on models which fall under the classification of Deep Learning. The advent of deep learning has dramatically improved the state-of-the-art in many common machine learning benchmarks. Some examples include classifying objects from images, language modeling, speech transcription, text-to-speech, and machine translation. In some tasks, the progress has been so remarkable that the performance of the models now matches or exceeds that of human experts.

Deep learning works well for the task of transcribing real-valued sequences as this framework falls into the regime of supervised learning. We put forth a framework which, when utilized well, we believe can solve many sequence transcription problems to near human expert levels. However, not all sequence transcription problems can be solved to a high degree of accuracy with today's algorithms and technology. Types of problems that can typically be solved with these methods include "perception" problems and more broadly speaking problems which an expert human can do with a few seconds or less of thought.[1] Both speech transcription, keyword spotting and detecting arrhythmias in electrocardiograms meet this criteria.

Traditional speech recognition systems rely on many hand-engineered components all brought together by the hidden Markov model framework. Similarly, prior work in arrhythmia detection also relies heavily on feature engineering and other rule-based algorithms for beat and rhythm detection. While these models and the assumptions they make can be useful to achieve generalization in low-resource settings, they also introduce bias which is difficult to overcome as we increase the available resources. They introduce a glass-ceiling on performance. Compared to these previous approaches for solving sequence transcription problems, deep learning is much more *end-to-end*. Deep learning based models can be trained end-to-end directly on the input and output pairs in the dataset or with minimal pre and post processing. These models are substantially more general and do not make many of the prior assumptions that come along with the hand-engineered components of traditional systems.

Since deep learning models are much more general, they can easily fail to perform well on unseen data. They require two ingredients to achieve low generalization error. First we must carefully tune the architecture and other constraints encoded in the model. Deep learning can be viewed as a framework for *differentiable computing*. This framework consists of differentiable modules (e.g. recurrent units, convolutional units, batch normalization, etc.) and a recipe for connecting these modules and efficiently

[1]Classifying problems which can be solved by deep learning in this way is due to Andrew Ng in e.g. https://hbr.org/2016/11/what-artificial-intelligence-can-and-cant-do-right-now.

learning their parameters. The building blocks are able to introduce useful prior information for the problem at hand while also maintaining flexibility since the model can learn to overcome these biases during the learning process. However, finding the right set of building blocks, sequencing them correctly and tuning their hyperparameters can require considerable computational resources and careful human effort. As we show in the following applications this effort can yield dramatic improvements to the model's ability to learn and generalize well.

The second critical ingredient is the amount and quality of annotated data we have available. Better models can compensate for less data and vice versa. However, in the following applications we achieve the best performance by jointly optimizing the two together. In the speech recognition applications, we construct datasets as large as 12,000 hours. Prior work on publicly available speech corpora typically used datasets in the 80-300 hour range. For the task of arrhythmia detection, we develop a corpus which contains more than 500 times the number of unique patients as the most commonly used publicly available benchmark.

In order to efficiently iterate over model architectures on large annotated datasets, we require considerable computation. The advent of GPUs in deep learning has unlocked what otherwise would be a computationally intractable problem. Even in the past few years, the performance of these models with off-the-shelf hardware and software has improved multiple orders of magnitude. Despite these dramatic improvements, in some cases considerable effort must still be spent making these models efficient to train and deploy. In particular, the scale of the speech transcription problem we study is such that data parallel optimization algorithms are essential. We develop and discuss techniques for efficient computation on the GPU, efficient I/O and network bandwidth utilization and efficient, low-latency deployments.

In summary, we develop end-to-end models for transcribing real-valued sequences and discuss three applications of these models. The first is detecting abnormal heart activity in electrocardiograms. The second is large vocabulary continuous speech recognition, and the third is keyword spotting and voice activity detection. In all cases we show how to scale high capacity models to unprecedentedly large datasets. With these techniques we can achieve performance comparable to that of human experts and state-of-the-art error rates in speech recognition for multiple languages.

1.1 Contributions per Chapter

We present the following contributions in the chapters of this dissertation:

Chapter 2: Background. This chapter provides some background on the models used throughout this work including RNNs and CTC. We start with a brief introduction to RNNs including LSTMs and GRUs. We also discuss various optimization algorithms used for RNNs as well as gradient clipping. We follow this with an in-depth look at CTC. We explain the CTC loss function and how to efficiently compute it as well as how to perform inference with a CTC trained model. The next section discusses properties and limitations of CTC and we follow this by placing CTC in context with HMMs and encoder-decoder models. We also include a short section on practical advice when using CTC. At the end of this section, we give a short bibliography and references for further study of the related material.

Chapter 3: Application to Arrhythmia Detection. In Chapter 3 we develop an algorithm which exceeds the performance of board certified cardiologists in detecting a wide range of heart arrhythmias from electrocardiograms recorded with a single-lead wearable monitor. We build a dataset with more than 500 times the number of unique patients than previously studied corpora. On this dataset, we train a 34-layer convolutional neural network which maps a sequence of ECG samples to a sequence of rhythm classes. Committees of board-certified cardiologists annotate a gold standard test set on which we compare the performance of our model to that of 6 other individual cardiologists. For ten arrhythmias, Sinus rhythm and noise the model achieves performance comparable to that of expert cardiologists when using a committee consensus label as the ground truth.

Chapter 4: First-pass Speech Recognition. Here we present a method to perform first-pass large vocabulary continuous speech recognition using only a neural network and language model. Deep neural network acoustic models are now commonplace in HMM-based speech recognition systems, but building such systems is a complex, domain-specific task. Prior work demonstrated the feasibility of discarding the HMM sequence modeling framework by directly predicting transcript text from audio. This chapter extends this approach in two ways. First, we demonstrate that a straightforward recurrent neural network architecture can achieve a high level of accuracy. Second, we propose and evaluate a modified prefix-search decoding algorithm. This approach to decoding enables first-pass speech recognition with a language model, completely unaided by the cumbersome infrastructure of HMM-based systems. Experiments on the Wall Street Journal corpus demonstrate fairly competitive word error rates, and the importance of bi-directional network recurrence.

Chapter 5: Deep Speech – Scaling Up End-to-end Speech Recognition. In Chapter 5, we present a state-of-the-art speech recognition system developed using end-to-end deep learning. Our architecture is significantly simpler than traditional speech systems, which rely on laboriously engineered processing pipelines; these traditional systems also tend to perform poorly when used in noisy environments. In contrast, our system does not need hand-designed components to model background noise, reverberation, or speaker variation, but instead directly learns a function that is robust to such effects. We do not need a phoneme dictionary, nor even the concept of a "phoneme." Key to our approach is a well-optimized RNN training system that uses multiple GPUs, as well as a set of novel data synthesis techniques that allow us to efficiently obtain a large amount of varied data for training. Our system, called Deep Speech, outperforms previously published results on the widely studied Switchboard Hub5'00, achieving 16.0% error on the full test set. Deep Speech also handles challenging noisy environments better than widely used, state-of-the-art commercial speech systems.

Chapter 6: Deep Speech 2 – End-to-End Speech Recognition in English and Mandarin. We show that an end-to-end deep learning approach can be used to recognize either English or Mandarin Chinese speechtwo vastly different languages. We show how to improve upon the Deep Speech model along three axis: the amount of annotated data, the architecture of the model and the efficiency of computation. Th application of HPC techniques, enables experiments that previously took weeks to now run in days. This allows us to iterate more quickly to identify superior architectures and algorithms. As a result, in several cases, our system is competitive with the transcription of human workers when benchmarked on standard datasets. Finally, using a technique called Batch Dispatch with GPUs in the data center, we show that our system can be inexpensively deployed in an online setting, delivering low latency when serving users at scale.

Chapter 7: Keyword Spotting and Voice Activity Detection. In this chapter we show how a single neural network architecture can be used for two tasks: online keyword spotting and voice activity detection. We develop novel inference algorithms for an end-to-end RNN trained with the CTC loss function which allow our model to achieve high accuracy on both keyword spotting and voice activity detection without retraining. In contrast to prior voice activity detection models, this architecture does not require aligned training data and uses the same parameters as the keyword spotting model. This allows us to deploy a high quality voice activity detector with no additional memory or maintenance requirements.

1.2 First Published Appearances

Most contributions described in this thesis have first appeared in various publications. Much of the background material on CTC in Chapter 2 first appeared in [108]. The results of Chapter 3 first appeared in [47]. The results on first-pass speech recognition in Chapter 4 were in [49]. Chapters 5 on Deep Speech and 6 on Deep Speech 2 were previously published in [48] and [2] respectively. Finally, Chapter 7 on keyword Spotting and voice activity detection was first published in [82].

Chapter 2

Background

2.1 Deep Neural Networks

In this work we primarily rely on deep neural networks (DNNs) for function approximation. A DNN is a multi-layered stack of differentiable computational building blocks. We use many standard building blocks such as convolutions, linear layers, and several nonlinear functions. In this section we give a brief introduction to recurrent neural networks (RNNs) and optimization techniques commonly used for RNNs. Other more novel building blocks such as recurrent batch normalization are introduced in the sections where they are used.

2.1.1 Recurrent Neural Networks

In order to incorporate an arbitrary amount of temporal context, we can use a recurrent neural network (RNN). An RNN computes the next step hidden state based on the current input and the previous hidden state:

$$h_t = f(x_t, h_{t-1}).$$

A vanilla RNN models f as

$$h_t = g(Wx_t + Uh_{t-1} + b)$$

where g is a nonlinearity such as the logistic sigmoid function, the hyperbolic tangent function or the rectified nonlinear unit. The variables W, U and b are tunable parameters.

One problem with vanilla RNNs is that the gradient can vanish over time causing learning long-term dependencies to become difficult. The Long Short-term Memory (LSTM) cell can avoid this. The LSTM computation is given by

$$\begin{aligned}
i_t &= \sigma(W_i x_t + U_i h_{t-1}) \\
f_t &= \sigma(W_f x_t + U_f h_{t-1}) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1}) \\
\tilde{c}_t &= g(W_c x_t + U_c h_{t-1}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ g(c_t)
\end{aligned}$$

Here g is any nonlinear activation function and σ is the logistic sigmoid function. Also, we use $\circ$ to denote element-wise multiplication. Note, we have left out the biases for simplicity; however, they are typically included in the first four equations.

Another type of RNN cell which works well in practice is the Gated Recurrent Unit (GRU). The GRU computation is given by

$$\begin{aligned}
r_t &= \sigma(W_r x_t + U_r h_{t-1}) \\
z_t &= \sigma(W_z x_t + U_z h_{t-1}) \\
\tilde{h}_t &= g(W_h x_t + r_t \circ U_h h_{t-1}) \\
h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t
\end{aligned}$$

An RNN can be either unidirectional or bidirectional. Unidirectional RNNs typically operate forwards in time only influencing the current hidden state with past information. Bidirectional RNNs also incorporate future information. The equations for a bidirectional RNN are

$$\begin{aligned}\overrightarrow{h}_t &= f(x_t, \overrightarrow{h}_{t-1}) \\ \overleftarrow{h}_t &= f(x_t, \overleftarrow{h}_{t-1}) \\ h_t &= \overrightarrow{h}_t + \overleftarrow{h}_t\end{aligned}$$

Here we sum the forward and backward hidden states, however, concatenating them is also common.

2.1.2 Optimization

Throughout this work and in almost all modern applications of DNNs a variation of stochastic gradient descent (SGD) is used to optimize the model's parameters. The basic SGD update is given by

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_\theta \mathcal{L}(X, Y)$$

where $\mathcal{L}(X, Y)$ is the loss function computed on the training pair (X, Y) and α_t is the learning rate used at time t. More commonly we use a *minibatch* of examples to compute each update as this is computationally much more efficient.

Momentum augments the plain SGD algorithm with a velocity term and can accelerate learning:

$$\begin{aligned}v_t &= \rho v_{t-1} + \alpha_t \nabla_\theta \mathcal{L}(X, Y) \\ \theta_t &= \theta_{t-1} - v_t\end{aligned}$$

Here the parameter ρ dictates the amount of momentum to use and is typically set in the range $[0.9, 0.99]$.

Nesterov's accelerated gradient (NAG) in some cases works better than classical momentum. The NAG update is given by

$$\begin{aligned}\tilde{\theta} &= \theta_t - \rho v_{t-1} \\ v_t &= \rho v_{t-1} + \alpha_t \nabla_{\tilde{\theta}} \mathcal{L}(X, Y) \\ \theta_t &= \theta_{t-1} - v_t\end{aligned}$$

Notice, the gradient is evaluated at $\tilde{\theta}$ which includes the velocity term. This update attempts to predict what the gradient is at the future location of the parameters after incorporating the velocity term.

There are many alternative updates most which attempt to incorporate some estimate of the variance or second order information of the gradient. Another alternative we use in this work is Adaptive Moment Estimation (Adam) [69].

Gradient Clipping

The gradient of an RNN can be unstable and in some cases explode to very large magnitudes. This tends to happen particularly on long sequences with unusual dynamics. Exploding gradients can result in unstable optimization which can diverge at any point in the learning process.

A very effective technique to counter this problem is gradient clipping. Gradient clipping simply scales the magnitude of the parameters to be less than some preset threshold

$$\hat{\theta} = \frac{\min\{\|\theta\|, \gamma\}}{\|\theta\|} \theta$$

where γ can be set in the range of one standard deviation above the mean gradient norm.

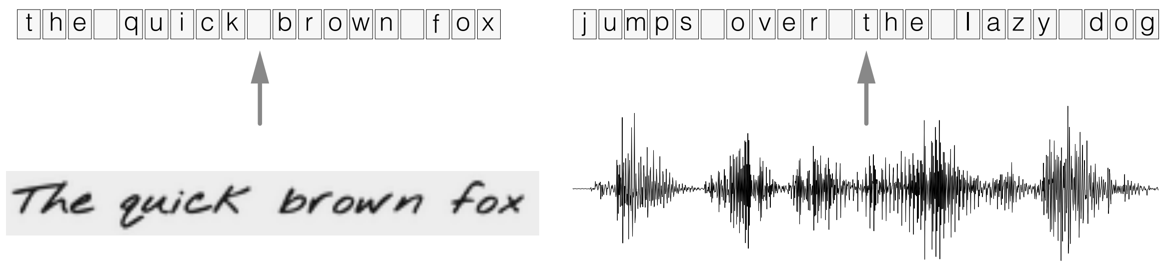

t h e q u i c k b r o w n f o x
j u m p s o v e r t h e l a z y d o g
The quick brown fox

Figure 2.1: An example of two sequence transcription problems: handwriting recognition (left) and speech recognition (right).

2.2 Connectionist Temporal Classification

In speech recognition, for example, we have a dataset of audio clips and corresponding transcripts. Unfortunately, we do not know how the characters in the transcript align to the audio. This makes training a speech recognizer harder than it might at first seem.

Without this alignment, the simple approaches are not available to us. We could devise a rule like "one character corresponds to ten inputs". However, people's rates of speech vary, so this type of rule can always be broken. Another alternative is to hand-align each character to its location in the audio. From a modeling standpoint this works well – we would know the ground truth for each input time-step. However, for any reasonably sized dataset this is prohibitively time consuming.

This problem is not specific to speech recognition. We see it in many other places. Handwriting recognition from images or sequences of pen strokes is one example. Action labelling in videos is another.

Connectionist Temporal Classification (CTC) is a way to get around not knowing the alignment between the input and the output. As we will see, it is especially well suited to applications like speech and handwriting recognition.

To be a bit more formal, consider mapping input sequences $X = [x_1, x_2, \ldots, x_T]$, such as audio, to corresponding output sequences $Y = [y_1, y_2, \ldots, y_U]$, such as transcripts. We want to find an accurate mapping from X's to Y's.

There are challenges which get in the way of us using simpler supervised learning algorithms. In particular:

- Both X and Y can vary in length.
- The ratio of the lengths of X and Y can vary.
- We don't have an accurate alignment (correspondence of the elements) of X and Y.

The CTC algorithm overcomes these challenges. For a given X it gives us an output distribution over all possible Y's. We can use this distribution either to *infer* a likely output or to assess the *probability* of a given output.

Not all ways of computing the loss function and performing inference are tractable. We will require that CTC do both of these efficiently.

Loss Function: For a given input, we would like to train our model to maximize the probability it assigns to the right answer. To do this, we will need to efficiently compute the conditional probability $p(Y \mid X)$. The function $p(Y \mid X)$ should also be differentiable, so we can use gradient descent.

Inference: Naturally, after training the model, we want to use it to infer a likely Y given an X. This means solving

$$Y^* \;=\; \underset{Y}{\operatorname{arg\,max}} \;\; p(Y \mid X).$$

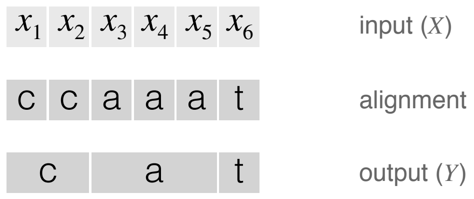
Ideally Y^* can be found efficiently. With CTC we will settle for an approximate solution that is not too expensive to find.

The CTC algorithm can assign a probability for any Y given an X. The key to computing this probability is how CTC thinks about alignments between inputs and outputs. We will start by looking at these alignments and then show how to use them to compute the loss function and perform inference.

2.2.1 Alignment

The CTC algorithm is *alignment-free* – it does not require an alignment between the input and the output. However, to get the probability of an output given an input, CTC works by summing over the probability of all possible alignments between the two. We need to understand what these alignments are in order to understand how the loss function is ultimately calculated.

To motivate the specific form of the CTC alignments, first consider a naive approach. As an example, assume the input has length six and $Y =$ [c, a, t]. One way to align X and Y is to assign an output character to each input step and collapse repeats.

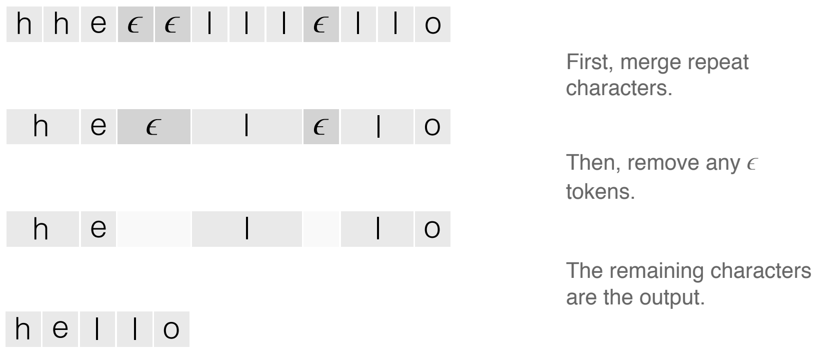


This approach has two problems.

- Often, it does not make sense to force every input step to align to some output. In speech recognition, for example, the input can have stretches of silence with no corresponding output.
- We have no way to produce outputs with multiple characters in a row. Consider the alignment [h, h, e, l, l, l, o]. Collapsing repeats will produce "helo" instead of "hello".

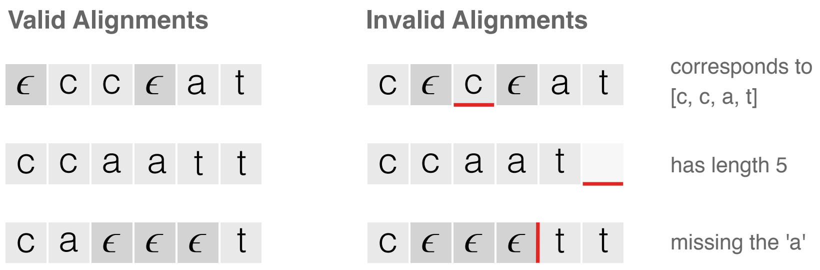
To get around these problems, CTC introduces a new token to the set of allowed outputs. This new token is sometimes called the *blank* token. We refer to it here as ϵ. The ϵ token does not correspond to anything and is simply removed from the output.

The alignments allowed by CTC are the same length as the input. We allow any alignment which maps to Y after merging repeats and removing ϵ tokens:


h h e ϵ ϵ l l l ϵ l l o
h e ϵ l ϵ l o
h e l l o
h e l l o
First, merge repeat characters.
Then, remove any ϵ tokens.
The remaining characters are the output.

If Y has two of the same character in a row, then a valid alignment must have an ϵ between them. With this rule in place, we can differentiate between alignments which collapse to "hello" and those which collapse to "helo".

Let's go back to the output [c, a, t] with an input of length six. Here are a few more examples of valid and invalid alignments.



The CTC alignments have a few notable properties. First, the allowed alignments between X and Y are monotonic. If we advance to the next input, we can keep the corresponding output the same or advance to the next one. A second property is that the alignment of X to Y is many-to-one. One or more input elements can align to a single output element but not vice-versa. This implies a third property: the length of Y cannot be greater than the length of X.

2.2.2 Loss Function

The CTC alignments give us a natural way to go from probabilities at each time-step to the probability of an output sequence. To be precise, the CTC objective for a single (X, Y) pair is:

$$p(Y \mid X) \quad = \sum_{A \in \mathcal{A}_{X,Y}} \prod_{t=1}^{T} p_t(a_t \mid X)$$

The CTC conditional probability marginalizes over the set of valid alignments computing the probability for a single alignment step-by-step.

Models trained with CTC typically use a recurrent neural network (RNN) to estimate the per time-step probabilities, $p_t(a_t \mid X)$. An RNN usually works well since it accounts for context in the input, but we are free to use any learning algorithm which produces a distribution over output classes given a fixed-size slice of the input.

If we are not careful, the CTC loss can be very expensive to compute. We could try the straightforward approach and compute the score for each alignment summing them all up as we go. The problem is there can be a massive number of alignments.[1] For most problems this would be too slow.

Thankfully, we can compute the loss much faster with a dynamic programming algorithm. The key insight is that if two alignments have reached the same output at the same step, then we can merge them.

Since we can have an ϵ before or after any token in Y, it is easier to describe the algorithm using a sequence which includes them. We will work with the sequence

$$Z \quad = \quad [\epsilon,\ y_1,\ \epsilon,\ y_2,\ \ldots,\ \epsilon,\ y_U,\ \epsilon]$$

which is Y with an ϵ at the beginning, end, and between every character.

Let α be the score of the merged alignments at a given node. More precisely, $\alpha_{s,t}$ is the CTC score of the subsequence $Z_{1:s}$ after t input steps. The final CTC score, $P(Y \mid X)$, is computed from from the α's at the last time-step. As long as we know the values of α at the previous time-step, we can compute $\alpha_{s,t}$. There are two

[1] For a Y of length U without any repeat characters and an X of length T the size of the set is $\binom{T+U}{T-U}$. For $T = 100$ and $U = 50$ this number is almost 10^{40}.

cases.

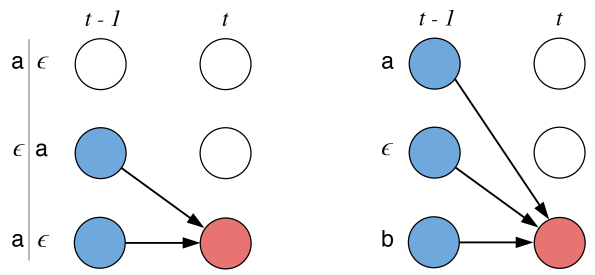


Figure 2.2: In Case 1 (left) we cannot skip the previous token whereas in Case 2 (right) we can.

Case 1: In this case, we cannot jump over z_{s-1}, the previous token in Z. The first reason is that the previous token can be an element of Y, and we cannot skip elements of Y. Since every element of Y in Z is followed by an ϵ, we can identify this when $z_s = \epsilon$. The second reason is that we must have an ϵ between repeat characters in Y. We can identify this when $z_s = z_{s-2}$.

To ensure we do not skip z_{s-1}, we can either be there at the previous time-step or have already passed through at some earlier time-step. As a result there are two positions we can transition from.

$$\alpha_{s,t} \;=\; (\alpha_{s-1,t-1} + \alpha_{s,t-1}) \cdot p_t(z_s \mid X)$$

In words, the probability of the subsequence $Z_{1:s}$ after t input steps is the sum of the probabilities of the two valid subsequences after $t-1$ input steps times the probability of the current character at input step t.

Case 2: In the second case, we are allowed to skip the previous token in Z. We have this case whenever z_{s-1} is an ϵ between unique characters. As a result there are three positions we could have come from at the previous step.

$$\alpha_{s,t} \;=\; (\alpha_{s-2,t-1} + \alpha_{s-1,t-1} + \alpha_{s,t-1}) \cdot p_t(z_s \mid X).$$

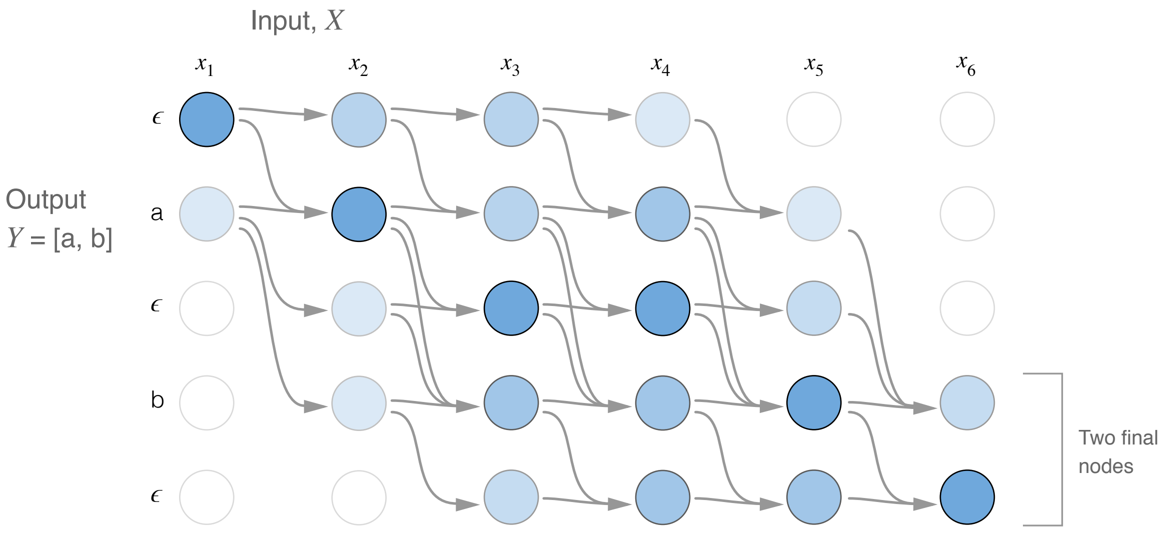


Figure 2.3: An example of the computation performed by CTC's dynamic programming algorithm. Every valid alignment has a path in this graph. Node (s, t) in the diagram represents $\alpha_{s,t}$ the CTC score of the subsequence $Z_{1:s}$ after t input steps.

There are two valid starting nodes and two valid final nodes since the ϵ at the beginning and end of the sequence is optional. The complete probability is the sum of the two final nodes.

Now that we can efficiently compute the loss function, the next step is to compute a gradient and train the model. The CTC loss function is differentiable with respect to the per time-step output probabilities since it is just sums and products of them. Given this, we can analytically compute the gradient of the loss function with respect to the (unnormalized) output probabilities and from there run backpropagation as usual.

For a training set $\mathcal{D}$, the model's parameters are tuned to minimize the negative log-likelihood

$$\sum_{(X,Y)\in\mathcal{D}} -\log\ p(Y \mid X)$$

instead of maximizing the likelihood directly.

2.2.3 Inference

After training the model, we would like to use it to find a likely output for a given input. More precisely, we need to solve:

$$Y^* \quad = \quad \underset{Y}{\operatorname{arg\,max}} \quad p(Y \mid X)$$

One heuristic is to take the most likely output at each time-step. This gives us the alignment with the highest probability:

$$A^* \quad = \quad \underset{A}{\operatorname{arg\,max}} \quad \prod_{t=1}^{T} p_t(a_t \mid X)$$

We can then collapse repeats and remove ϵ tokens to get Y.

For many applications this heuristic works well, especially when most of the probability mass is alloted to a single alignment. However, this approach can sometimes miss easy to find outputs with much higher probability. The problem is, it does not take into account the fact that a single output can have many alignments.

As an example, assume the alignments [a, a, ϵ] and [a, a, a] individually have lower probability than [b, b, b]. But the sum of their probabilities is actually greater than that of [b, b, b]. The naive heuristic will incorrectly propose Y = [b] as the most likely hypothesis. It should have chosen Y = [a]. To fix this, the algorithm needs to account for the fact that [a, a, a] and [a, a, ϵ] collapse to the same output.

We can use a modified beam search to solve this. Given limited computation, the modified beam search will not necessarily find the most likely Y. It does, at least, have the nice property that we can trade-off more computation (a larger beam-size) for an asymptotically better solution.

A regular beam search computes a new set of hypotheses at each input step. The new set of hypotheses is generated from the previous set by extending each hypothesis with all possible output characters and keeping only the top candidates.

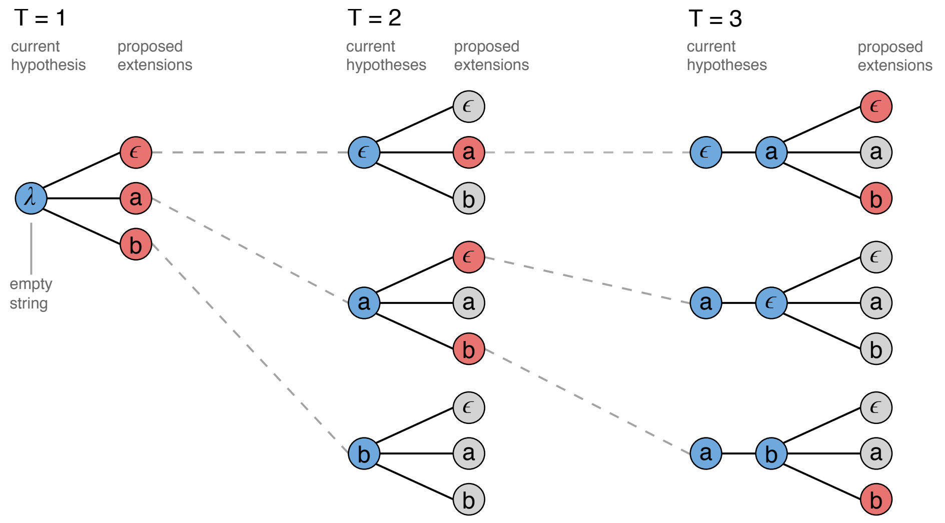


Figure 2.4: A standard beam search algorithm with an alphabet of $\{\epsilon, a, b\}$ and a beam size of three.

We can modify the vanilla beam search to handle multiple alignments mapping to the same output. In this case instead of keeping a list of alignments in the beam, we store the output prefixes after collapsing repeats and removing ϵ characters. At each step of the search we accumulate scores for a given prefix based on all the alignments which map to it.

A proposed extension can map to two output prefixes if the character is a repeat. This is shown at $T = 3$ in the figure above where ‘a is proposed as an extension to the prefix [a]. Both [a] and [a, a] are valid outputs for this proposed extension.

When we extend [a] to produce [a, a], we only want include the part of the previous score for alignments which end in ϵ. Remember, the ϵ is required between repeat characters. Similarly, when we do not extend the prefix and produce [a], we should only include the part of the previous score for alignments which do not end in ϵ.

Given this, we have to keep track of two probabilities for each prefix in the beam. The probability of all alignments which end in ϵ and the probability of all alignments which do not end in ϵ. When we rank the hypotheses at each step before pruning the beam, we will use their combined scores.

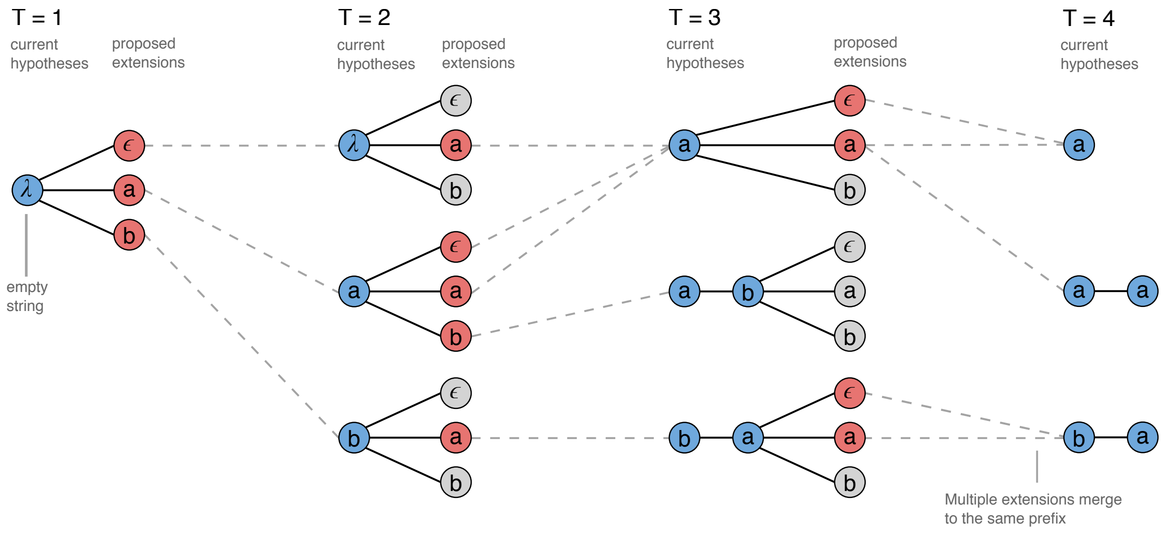


Figure 2.5: The CTC beam search algorithm with an output alphabet $\{\epsilon, a, b\}$ and a beam size of three.

In some problems, such as speech recognition, incorporating a language model over the outputs significantly improves accuracy. We can include the language model as a factor in the inference problem.

$$Y^* \;=\; \operatorname*{arg\,max}_{Y} \;\; p(Y \mid X) \cdot p(Y)^{\alpha} \cdot L(Y)^{\beta}$$

The terms are respectively the CTC conditional probability, the language model probability and a "word" insertion bonus. The function $L(Y)$ computes the length of Y in terms of the language model tokens and acts as a word insertion bonus. With a word-based language model $L(Y)$ counts the number of words in Y. If we use a character-based language model then $L(Y)$ counts the number of characters in Y. The language model scores are only included when a prefix is extended by a character (or word) and not at every step of the algorithm. This causes the search to favor shorter prefixes, as measured by $L(Y)$, since they do not include as many language model updates. The word insertion bonus helps with this. The parameters α and β are usually set by cross-validation.

The language model scores and word insertion term can be included in the beam search. Whenever we propose to extend a prefix by a character, we can include the language model score for the new character given the prefix so far.

The implementation of this decoding algorithm is dense and tricky to get right. See Algorithm 1 for a more detailed description.

2.2.4 Properties of CTC

We mentioned a few important properties of CTC so far. Here we will go into more depth on what these properties are and what trade-offs they offer.

Conditional Independence

One of the most commonly cited shortcomings of CTC is the conditional independence assumption it makes. The model assumes that every output is conditionally independent of the other outputs given the input. This is a bad assumption for many sequence to sequence problems.

Say we had an audio clip of someone saying “triple A” [12]. Another valid transcription could be “AAA”. If the first letter of the predicted transcription is ‘A’, then the next letter should be ‘A’ with high probability and ‘r’ with low probability. The conditional independence assumption does not allow for this.

In fact speech recognizers using CTC do not learn a language model over the output nearly as well as models which are conditionally dependent [8]. However, a separate language model can be included and usually gives a good boost to accuracy.

The conditional independence assumption made by CTC is not always a bad thing. Baking in strong beliefs over output interactions makes the model less adaptable to new or altered domains. For example, we might want to use a speech recognizer trained on phone conversations between friends to transcribe customer support calls. The language in the two domains can be quite different even if the acoustic model is similar. With a CTC acoustic model, we can easily swap in a new language model as we change domains.

Alignment Properties

The CTC algorithm is *alignment-free*. The objective function marginalizes over all alignments. While CTC does make strong assumptions about the form of alignments between X and Y, the model is agnostic as to how probability is distributed amongst them. In some problems CTC ends up allocating most of the probability to a single alignment. However, this is not guaranteed.[2]

As mentioned before, CTC only allows *monotonic* alignments. In problems such as speech recognition this may be a valid assumption. For other problems like machine translation where a future word in a target sentence can align to an earlier part of the source sentence, this assumption is a deal-breaker.

Another important property of CTC alignments is that they are *many-to-one*. Multiple inputs can align to at most one output. In some cases this may not be desirable. We might want to enforce a strict one-to-one correspondence between elements of X and Y. Alternatively, we may want to allow multiple output elements to align to a single input element. For example, the characters "th" might align to a single input step of audio. A character based CTC model would not allow that.

The many-to-one property implies that the output cannot have more time-steps than the input.[3] This is usually not a problem for speech and handwriting recognition since the input is much longer than the output. However, for other problems where Y is often longer than X, CTC will not work.

[2]We could force the model to choose a single alignment by replacing the sum with a max in the objective function,

$$p(Y \mid X) \;=\; \max_{A \in \mathcal{A}_{X,Y}} \prod_{t=1}^{T} p(a_t \mid X).$$

[3]If Y has r consecutive repeats, then the length of Y must be less than the length of X by $2r-1$.

2.2.5 CTC In Context

In this section we will discuss how CTC relates to other commonly used algorithms for sequence modeling.

HMMs

At a first glance, a Hidden Markov Model (HMM) seems quite different from CTC. But, the two algorithms are actually quite similar. Understanding the relationship between them will help us understand what advantages CTC has over HMM sequence models and give us insight into how CTC could be changed for various use cases.

Using the same notation as before, let X be the input sequence and Y be the output sequence with lengths T and U respectively. We are interested in learning $p(Y \mid X)$. One way to simplify the problem is to apply Bayes' Rule:

$$p(Y \mid X) \;\propto\; p(X \mid Y)\, p(Y).$$

The $p(Y)$ term can be any language model, so let us focus on $p(X \mid Y)$. Like before we will let $\mathcal{A}$ be a set of allowed of alignments between X and Y. Members of $\mathcal{A}$ have length T. Let us otherwise leave $\mathcal{A}$ unspecified for now; we will come back to it later. We can marginalize over alignments to get

$$p(X \mid Y) \;=\; \sum_{A \in \mathcal{A}} p(X, A \mid Y).$$

To simplify notation, we remove the conditioning on Y, it will be present in every $p(\cdot)$. With two assumptions we can write down the standard HMM.

$$p(X) \;\;=\;\; \sum_{A \in \mathcal{A}} \prod_{t=1}^{T} \; p(x_t \mid a_t) \cdot p(a_t \mid a_{t-1})$$

The probability of the input marginalizes over alignments the emission probability,

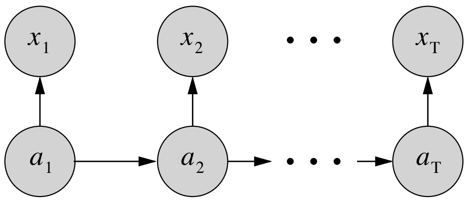


Figure 2.6: The graphical model for an HMM.

$p(x_t \mid a_t)$, and the transition probability, $p(a_t \mid a_{t-1})$. The first assumption is the usual Markov property. The state a_t is conditionally independent of all historic states given the previous state a_{t-1}. The second is that the observation x_t is conditionally independent of everything given the current state a_t.

Now we can take just a few steps to transform the HMM into CTC and see how the two models relate. First, assume that the transition probabilities $p(a_t \mid a_{t-1})$ are uniform. This gives

$$p(X) \;\propto\; \sum_{A \in \mathcal{A}} \prod_{t=1}^{T} p(x_t \mid a_t).$$

There are only two differences from this equation and the CTC loss function. The first is that we are learning a model of X given Y as opposed to Y given X. The second is how the set $\mathcal{A}$ is produced. We deal with each in turn.

The HMM can be used with discriminative models which estimate $p(a \mid x)$. To do this, we apply Bayes' Rule and rewrite the model as

$$\begin{aligned} p(X) \;&\propto\; \sum_{A \in \mathcal{A}} \prod_{t=1}^{T} \frac{p(a_t \mid x_t)\, p(x_t)}{p(a_t)} \\ &\propto\; \sum_{A \in \mathcal{A}} \prod_{t=1}^{T} \frac{p(a_t \mid x_t)}{p(a_t)}. \end{aligned}$$

If we assume a uniform prior over the states a and condition on all of X instead of a

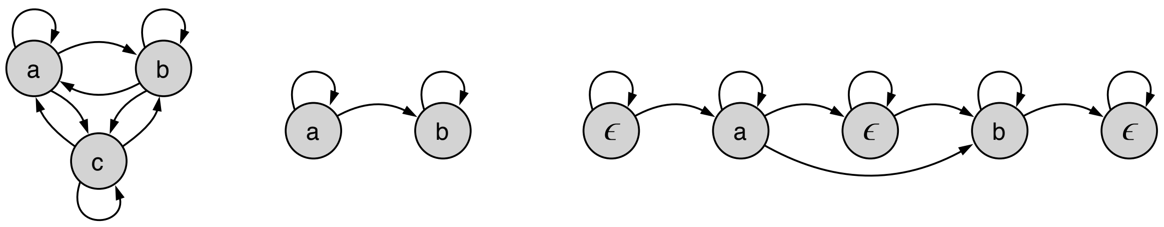


Figure 2.7: Different HMM architectures.

single element at a time, we arrive at

$$p(X) \;\propto\; \sum_{A \in \mathcal{A}} \prod_{t=1}^{T} p(a_t \mid X).$$

The above equation is essentially the CTC loss function, assuming the set $\mathcal{A}$ is the same. In fact, the HMM framework does not specify what $\mathcal{A}$ should consist of. This part of the model can be designed on a per-problem basis. In many cases the model does not condition on Y and the set $\mathcal{A}$ consists of all possible length T sequences from the output alphabet. In this case, the HMM can be drawn as an *ergodic* state transition diagram in which every state connects to every other state. The figure below shows this model with the alphabet or set of unique hidden states as $\{a, b, c\}$.

In our case the transitions allowed by the model are strongly related to Y. We want the HMM to reflect this. One possible model could be a simple linear state transition diagram. The figure below shows this with the same alphabet as before and Y = [a, b]. Another commonly used model is the *Bakis* or left-right HMM. In this model any transition which proceeds from the left to the right is allowed.

In CTC we augment the alphabet with ϵ and the HMM model allows a subset of the left-right transitions. The CTC HMM has two start states and two accepting states.

One possible source of confusion is that the HMM model differs for any unique Y.

This is in fact standard in applications such as speech recognition. The state diagram changes based on the output Y. However, the functions which estimate the observation and transition probabilities are shared.

Let's discuss how CTC improves on the original HMM model. First, we can think of the CTC state diagram as a special case HMM which works well for many problems of interest. Incorporating the blank as a hidden state in the HMM allows us to use the alphabet of Y as the other hidden states. This model also gives a set of allowed alignments which may be a good prior for some problems.

Perhaps most importantly, CTC is discriminative. It models $p(Y \mid X)$ directly, an idea that has been important in the past with other discriminative improvements to HMMs [137]. Discriminative training allows us to apply powerful learning algorithms like the RNN directly towards solving the problem we care about.

Encoder-Decoder Models

The encoder-decoder is perhaps the most commonly used framework for sequence modeling with neural networks. These models have an encoder and a decoder. The encoder maps the input sequence X into a hidden representation. The decoder consumes the hidden representation and produces a distribution over the outputs. We can write this as

$$\begin{aligned} H &= \mathsf{encode}(X) \\ p(Y \mid X) &= \mathsf{decode}(H). \end{aligned}$$

The $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$ functions are typically RNNs. The decoder can optionally be equipped with an attention mechanism. The hidden state sequence H has the same number of time-steps as the input, T. Sometimes the encoder subsamples the input. If the encoder subsamples the input by a factor s then H will have T/s time-steps.

We can interpret CTC in the encoder-decoder framework. This is helpful to understand the developments in encoder-decoder models that are applicable to CTC and to develop a common language for the properties of these models.

Encoder: The encoder of a CTC model can be just about any encoder we find in commonly used encoder-decoder models. For example the encoder could be a multi-layer bidirectional RNN or a convolutional network. There is a constraint on the CTC encoder that does not apply to the others. The input length cannot be sub-sampled so much that T/s is less than the length of the output.

Decoder: We can view the decoder of a CTC model as a simple linear transformation followed by a softmax normalization. This layer should project all T steps of the encoder output H into the dimensionality of the output alphabet.

We mentioned earlier that CTC makes a conditional independence assumption over the characters in the output sequence. This is one of the big advantages that other encoder-decoder models have over CTC – they can model the dependence over the outputs. However in practice, CTC is still more commonly used in tasks like speech recognition as we can partially overcome the conditional independence assumption by including an external language model.

2.2.6 Practitioner's Guide

So far we have mostly developed a conceptual understanding of CTC. Here we will go through a few implementation tips for practitioners.

Software: Even with a solid understanding of CTC, the implementation is difficult. The algorithm has several edge cases and a fast implementation should be written in a lower-level programming language. Open-source software tools make it much easier to get started:

- Baidu Research has open-sourced `warp-ctc`[4]. The package is written in C++

[4]https://github.com/baidu-research/warp-ctc

and CUDA. The CTC loss function runs on either the CPU or the GPU. Bindings are available for Torch, TensorFlow and PyTorch.

- TensorFlow has built in CTC loss and beam search functions for the CPU.[5]
- Nvidia also provides a GPU implementation of CTC in cuDNN versions 7 and up.[6]

Numerical Stability: Computing the CTC loss naively is numerically unstable. One method to avoid this is to normalize the αs at each time-step. The original publication has more detail on this including the adjustments to the gradient [40]. In practice this works well enough for medium length sequences but can still underflow for long sequences. A better solution is to compute the loss function in log-space with the log-sum-exp trick.[7]

Inference should also be done in log-space using the log-sum-exp trick.

Beam Search: There are a couple of good tips to know about when implementing and using the CTC beam search.

The correctness of the beam search can be tested as follows.

1. Run the beam search algorithm on an arbitrary input.
2. Save the inferred output $\bar{Y}$ and the corresponding score $\bar{c}$.
3. Compute the actual CTC score c for $\bar{Y}$.
4. Check that $\bar{c} \approx c$ with the former being no greater than the later. As the beam size increases the inferred output $\bar{Y}$ may change, but the two numbers should grow closer.

[5]https://www.tensorflow.org/api_docs/python/tf/nn

[6]https://developer.nvidia.com/cudnn

[7]When computing the sum of two probabilities in log space use the identity

$$\log(e^a + e^b) = \max\{a, b\} + \log(1 + e^{-|a-b|})$$

Most programming languages have a stable function to compute $\log(1 + x)$ when x is close to zero.

A common question when using a beam search decoder is the size of the beam to use. There is a trade-off between accuracy and runtime. We can check if the beam size is in a good range. To do this first compute the CTC score for the inferred output c_i. Then compute the CTC score for the ground truth output c_g. If the two outputs are not the same, we should have $c_g < c_i$. If $c_i << c_g$ then the ground truth output actually has a higher probability under the model and the beam search failed to find it. In this case a large increase to the beam size may be warranted.

2.3 Bibliographic Notes

Convolutional neural networks were popularized following the seminal work of Lecun, et al. in 1998 [79]. The LSTM RNN was proposed in 1997 [57] and the GRU was proposed in 2014 [16]. Olah gives a great introduction to RNNs and the LSTM cell [100]. Gradient clipping was first proposed by Pascanu, et al. in 2013 [103].

The CTC algorithm was first published by Graves et al. in 2006 [40]. The first experiments were on TIMIT, a popular phoneme recognition benchmark [86]. Chapter 7 of Graves' thesis [43] also gives a detailed treatment of CTC.

One of the first applications of CTC to large vocabulary speech recognition was by Graves et al. in 2014 [41]. They combined a hybrid DNN-HMM and a CTC trained model to achieve state-of-the-art results. Hannun et al. subsequently demonstrated state-of-the-art CTC based speech recognition on larger benchmarks [48]. A CTC model outperformed other methods on an online handwriting recognition benchmark in 2007 [85].

CTC has been used successfully in many other problems. Some examples are lip-reading from video [4], action recognition from video [58] and keyword detection in audio [36, 82].

Many extensions and improvements to CTC have been proposed. Here are a few. The *Sequence Transducer* discards the conditional independence assumption made by CTC [42]. As a consequence, the model allows the output to be longer than the input. The *Gram-CTC* model generalizes CTC to marginalize over n-gram output classes [84]. Other works have generalized CTC or proposed similar algorithms to account for segmental structure in the output [134, 73].

The Hidden Markov Model was developed in the 1960's with the first application to speech recognition in the 1970's. For an introduction to the HMM and applications to speech recognition see Rabiner's canonical tutorial [106].

Encoder-decoder models were developed in 2014 [16, 128]. The online publication *Distill* has an in-depth guide to attention in encoder-decoder models [99].

Chapter 3

Application to Arrhythmia Detection

3.1 Introduction

We develop a model which can diagnose irregular heart rhythms, also known as arrhythmias, from single-lead ECG signals better than a cardiologist. Key to exceeding expert performance is a deep convolutional network which can map a sequence of ECG samples to a sequence of arrhythmia annotations along with a novel dataset two orders of magnitude larger than previous datasets of its kind.

Many heart diseases, including Myocardial Infarction, AV Block, Ventricular Tachycardia and Atrial Fibrillation can all be diagnosed from ECG signals with an estimated 300 million ECGs recorded annually [54]. We investigate the task of arrhythmia detection from the ECG record. This is known to be a challenging task for computers but can usually be determined by an expert from a single, well-placed lead.

Arrhythmia detection from ECG recordings is usually performed by expert technicians and cardiologists given the high error rates of computerized interpretation. One study found that of all the computer predictions for non-sinus rhythms, only about 50% were correct [122]; in another study, only 1 out of every 7 presentations of second degree AV block were correctly recognized by the algorithm [45]. To automatically detect heart arrhythmias in an ECG, an algorithm must implicitly recognize the distinct wave types and discern the complex relationships between them over time. This is difficult due to the variability in wave morphology between patients as well as the presence of noise.

We train a 34-layer convolutional neural network (CNN) to detect arrhythmias in arbitrary length ECG time-series. In addition to classifying noise and the sinus rhythm, the network learns to classify and segment ten arrhythmia types present in the time-series. The model is trained end-to-end on a single-lead ECG signal sampled at 200Hz and a sequence of annotations for every second of the ECG as supervision. To make the optimization of such a deep model tractable, we use residual connections and batch-normalization [51, 61]. The depth increases both the non-linearity of the computation as well as the size of the context window for each classification decision.

We construct a dataset 500 times larger than other datasets of its kind [95, 39]. One of the most popular previous datasets, the MIT-BIH corpus contains ECG recordings from 47 unique patients. In contrast, we collect and annotate a dataset of about 54,000 unique patients from a pool of nearly 300,000 patients who have used the Zio Patch monitor[1] [131]. We intentionally select patients exhibiting abnormal rhythms in order to make the class balance of the dataset more even and thus the likelihood of observing unusual heart-activity high.

We test our model against board-certified cardiologists. A committee of three cardiologists serve as gold-standard annotators for the 336 examples in the test set. Our model achieves an AUC of greater than 0.90 for each of the 12 rhythm diagnoses, with an average AUC of 0.97. Average F1 scores for the model (0.81) exceeded those for averaged cardiologists (0.78).

[1]iRhythm Technologies, San Francisco, California

3.2 Related Work

Automatic high-accuracy methods for R-peak extraction have existed at least since the mid 1980's [101]. Current algorithms for R-peak extraction tend to use wavelet transformations to compute features from the raw ECG followed by finely-tuned threshold based classifiers [83, 90]. Because accurate estimates of heart rate and heart rate variability can be extracted from R-peak features, feature-engineered algorithms are often used for coarse-grained heart rhythm classification, including detecting tachycardias (fast heart rate), bradycardias (slow heart rate), and irregular rhythms. However, such features alone are not sufficient to distinguish between most heart arrhythmias since features based on the atrial activity of the heart as well as other features pertaining to the QRS morphology are needed.

Much work has been done to automate the extraction of other features from the ECG. For example, beat classification is a common sub-problem of heart-arrhythmia classification. Drawing inspiration from automatic speech recognition, Hidden Markov models with Gaussian observation probability distributions have been applied to the task of beat detection [24]. Artificial neural networks have also been used for the task of beat detection [91]. While these models have achieved high-accuracy for some beat types, they are not yet sufficient for high-accuracy heart arrhythmia classification and segmentation. For example, [3] train a neural network to distinguish between Atrial Fibrillation and Sinus Rhythm on the MIT-BIH dataset. While the network can distinguish between these two classes with high-accuracy, it does not generalize to noisier single-lead recordings or classify among the full range of 15 rhythms available in MIT-BIH. This is in part due to insufficient training data, and because the model also discards critical information in the feature extraction stage.

The most common dataset used to design and evaluate ECG algorithms is the MIT-BIH arrhythmia database [95] which consists of 48 half-hour strips of ECG data. Other commonly used datasets include the MIT-BIH Atrial Fibrillation dataset [94] and the QT dataset [75]. While useful benchmarks for R-peak extraction and beat-level annotations, these datasets are too small for fine-grained arrhythmia classification. The number of unique patients is in the single digit hundreds or fewer for these benchmarks. A recently released dataset captured from the AliveCor ECG monitor contains about 7000 records [23]. These records only have annotations for Atrial Fibrillation; all other arrhythmias are grouped into a single bucket. The dataset we develop contains 29,163 unique patients and 14 classes with hundreds of unique examples for the rarest arrhythmias.

Machine learning models based on deep neural networks have consistently been able to approach and often exceed human agreement rates when large annotated datasets are available [2, 138, 50]. These approaches have also proven to be effective in healthcare applications, particularly in medical imaging where pretrained ImageNet models can be applied [35, 46]. We draw on work in automatic speech recognition for processing time-series with deep convolutional neural networks and recurrent neural networks [48, 114], and techniques in deep learning to make the optimization of these models tractable [51, 52, 61].

3.3 Model

3.3.1 Problem Formulation

The ECG arrhythmia detection task is a sequence-to-sequence task which takes as input an ECG signal $X = [x_1, ..x_T]$, and outputs a sequence of labels $Y = [y_1, ...y_U]$, such that each y_u can take on one of 12 different rhythm classes. Each output label corresponds to a segment of the input. Together the output labels cover the full sequence.

For a single example in the training set, we optimize the cross-entropy objective function

$$\mathcal{L}(X, Y) = \frac{1}{U} \sum_{u=1}^{U} \log p(y_u \mid X)$$

where $p(\cdot)$ is the probability that the network assigns to the u-th output taking on the value y_u.

3.3.2 Model Architecture and Training

We use a convolutional neural network for the sequence-to-sequence learning task. The high-level architecture of the network is shown in Figure 3.1. The network takes as input a time-series of raw ECG signal, and outputs a sequence of label predictions. The 30 second long ECG signal is sampled at 200Hz, and the model outputs a new prediction once every 256 samples. After careful architectur search, arrive at a model which is 33 layers of convolution followed by a fully connected layer and a softmax.

In order to make the optimization of such a network tractable, we employ shortcut connections in a similar manner to those found in the Residual Network architecture [52]. The shortcut connections between neural-network layers optimize training by allowing information to propagate well in very deep neural networks. Before the input is fed into the network, it is normalized using a robust normalization strategy. The network consists of 16 residual blocks with 2 convolutional layers per block. The convolutional layers all have a filter length of 16 and have $64k$ filters, where k starts out as 1 and is incremented every 4-th residual block. Every alternate residual block subsamples its inputs by a factor of 2, thus the original input is ultimately subsampled by a factor of 2^8. When a residual block subsamples the input, the corresponding shortcut connections also subsample their input using a Max Pooling operation with the same subsample factor.

Before each convolutional layer we apply Batch Normalization [61] and a rectified linear activation, adopting the pre-activation block design [51]. The first and last

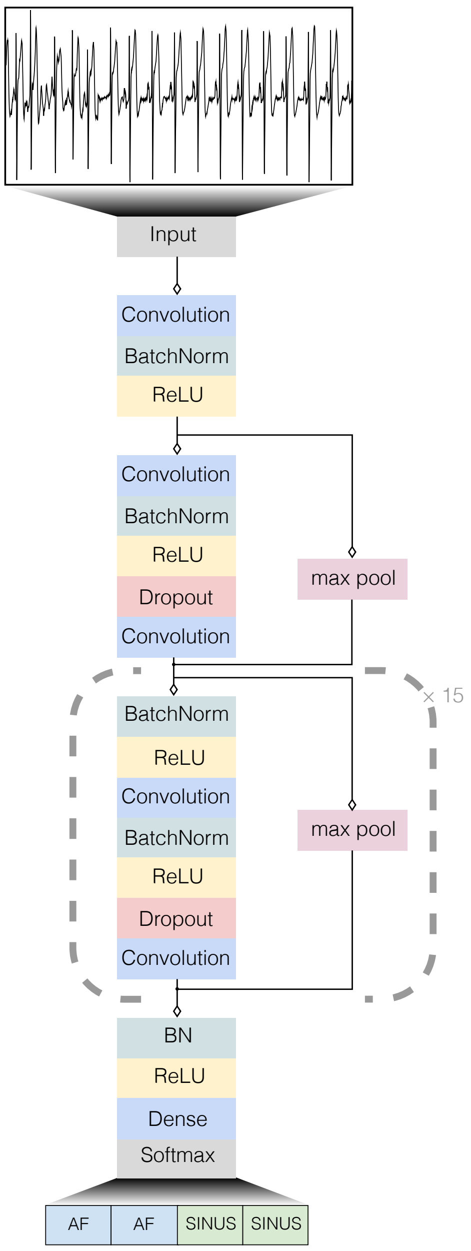


Figure 3.1: The architecture of the deep neural network consists of 33 convolutional layers followed by a fully-connected layer and a softmax layer.

layers of the network are special-cased due to this pre-activation block structure. We also apply Dropout [126] between the convolutional layers and after the non-linearity. The final fully connected layer and softmax activation produce a distribution over the 12 output classes for each time-step.

We train the networks from scratch, initializing the weights of the convolutional layers as in [50]. We use the Adam [69] optimizer with the default parameters and reduce the learning rate by a factor of 10 when the validation loss stops improving. We save the best model as evaluated on the validation set during the optimization process.

3.4 Datasets

3.4.1 Training

We collect and annotate a dataset of 91,232 ECG records from 53,877 patients. The ECG data is sampled at a frequency of 200 Hz and is collected from a single-lead, noninvasive and continuous monitoring device called the Zio Patch which has a wear period up to 14 days [131]. Each ECG record in the training set is 30 seconds long and can contain more than one rhythm type. Each record is annotated by a clinical ECG expert: the expert highlights segments of the signal and marks it as corresponding to one of the 12 rhythm classes.

The 30 second records were annotated using a web-based ECG annotation tool designed for this work. Label annotations were done by a group of Certified Cardiographic Technicians who have completed extensive training in arrhythmia detection and a cardiographic certification examination by Cardiovascular Credentialing International. The technicians were guided through the interface before they could annotate records. All rhythms present in a strip were labeled from their corresponding onset to offset, resulting in full segmentation of the input ECG data. To improve labeling consistency among different annotators, specific rules were devised regarding each rhythm transition.

We split the dataset into a training and validation set. The training set contains 90% of the data. We split the dataset so that there is no patient overlap between the training and validation sets (as well as the test set described below).

3.4.2 Testing

We collect a test set of 336 records from 328 unique patients. For the test set, ground truth annotations for each record were obtained by a committee of three board-certified cardiologists; there are three committees responsible for different splits of the test set. The cardiologists discussed each individual record as a group and came to a consensus labeling. For each record in the test set we also collect 6 individual annotations from cardiologists not participating in the group. This is used to assess performance of the model compared to an individual cardiologist.

3.4.3 Rhythm Classes

We identify 10 heart arrhythmias, sinus rhythm and noise for a total of 12 output classes. The arrhythmias are characterized by a variety of features. The noise label is assigned when the device is disconnected from the skin or when the baseline noise in the ECG makes identification of the underlying rhythm impossible.

The morphology of the ECG during a single heart-beat as well as the pattern of the activity of the heart over time determine the underlying rhythm. In some cases the distinction between the rhythms can be subtle yet critical for treatment. For example two forms of second degree AV Block, Mobitz I (Wenckebach) and Mobitz II (here included in the AVB category) can be difficult to distinguish. Wenckebach is considered benign and Mobitz II is considered pathological, requiring immediate attention [32].

3.5 Experimental Results

Evaluation Metrics

We use two metrics to measure model accuracy, using the cardiologist committee annotations as the ground truth.

Sequence Level Accuracy (F1): We measure the average overlap between the prediction and the ground truth sequence labels. For every record, a model is required to make a prediction approximately once per second (every 256 samples). These predictions are compared against the ground truth annotation.

Set Level Accuracy (F1): Instead of treating the labels for a record as a sequence, we consider the set of unique arrhythmias present in each 30 second record as the ground truth annotation. Set Level Accuracy, unlike Sequence Level Accuracy, does not penalize for time-misalignment within a record. We report the F1 score between the unique class labels from the ground truth and those from the model prediction.

In both the Sequence and the Set case, we compute the metrics for each class separately. We then compute aggregate results for AUC and F1 as the class-frequency weighted mean.

Model and Cardiologist Performance

We assess the cardiologist performance on the test set. Recall that each of the records in the test set has a ground truth label from a committee of three cardiologists as well as individual labels from a disjoint set of 6 other cardiologists. To assess cardiologist performance for each class, we take the average of all the individual cardiologist scores using the group label as the ground truth annotation.

Table 3.1 gives the per class and aggregate AUC scores. Our model achieves an AUC

	Sequence AUC	Set AUC
Atrial Fibrillation & Flutter	0.975	0.959
Atrio-ventricular Block	0.989	0.981
Bigeminy	0.998	0.997
Ectopic Atrial Rhythm	0.908	0.935
Idioventricular Rhythm	0.995	0.986
Junctional Rhythm	0.985	0.980
Noise	0.985	0.958
Sinus Rhythm	0.976	0.981
Supraventricular Tachycardia	0.972	0.940
Trigeminy	0.999	0.997
Ventricular Tachycardia	0.995	0.981
Wenckebach	0.982	0.981
Average	0.979	0.974

Table 3.1: The model AUC scores for each rhythm class and in aggregate.

of greater than 0.90 for each of the 12 rhythm diagnoses, with an average AUC of 0.97. We also compute the sensitivity at a specifity of 0.9 and vice versa in 3.2.

Table 3.3 shows the breakdown of both cardiologist and model sequence and set F1 across the different rhythm classes. The model outperforms the average cardiologist performance on most rhythms, noticeably outperforming the cardiologists in the AV Block set of arrhythmias which includes Mobitz I (Wenckebach), Mobitz II and complete heart block (both categorized as Atrio-ventricular Block). This is especially useful given the severity of second and third degree heart block and the importance of distinguishing these two from Wenckebach which is usually considered benign. The model also outperforms the cardiologist average accross all rhythm classes for both the sequence and set F1 score.

Figures 3.2 and 3.3 show the models performance at various operating thresholds on an ROC and precision-recall curve respectively. We show here three arrhythmias taken from one-third of the test set. We also compute and plot the operating point for the six individual cardiologists who annotated that third of the test set. The model outperforms almost all of the individual cardiologists. We see the same behaviour for

	Sensitivity	Specificity
Atrial Fibrillation & Flutter	0.923	0.914
Atrio-ventricular Block	0.991	0.964
Bigeminy	1.000	0.997
Ectopic Atrial Rhythm	0.754	0.665
Idioventricular Rhythm	0.990	0.990
Junctional Rhythm	0.982	0.959
Noise	0.965	0.968
Sinus Rhythm	0.934	0.946
Supraventricular Tachycardia	0.958	0.925
Trigeminy	1.000	0.999
Ventricular Tachycardia	1.000	0.981
Wenckebach	0.977	0.956

Table 3.2: The maximum model sensitivity with specificity greater than 0.9 and vice versa.

	Sequence F1		Set F1	
	Model	Cardiol.	Model	Cardiol.
Atrial Fibrillation & Flutter	0.802	0.679	0.817	0.687
Atrio-ventricular Block	0.850	0.769	0.830	0.756
Bigeminy	0.896	0.837	0.870	0.849
Ectopic Atrial Rhythm	0.537	0.476	0.545	0.529
Idioventricular Rhythm	0.751	0.632	0.818	0.720
Junctional Rhythm	0.640	0.684	0.778	0.674
Noise	0.852	0.768	0.704	0.689
Sinus Rhythm	0.886	0.847	0.934	0.907
Supraventricular Tachycardia	0.458	0.449	0.630	0.556
Trigeminy	0.909	0.843	0.870	0.816
Ventricular Tachycardia	0.520	0.566	0.653	0.769
Wenckebach	0.714	0.593	0.806	0.736
Average	0.808	0.750	0.809	0.778

Table 3.3: The F1 score for the sequence and set-level metrics comparing the model and the average of six individual cardiologist to the committee consensus ground truth.

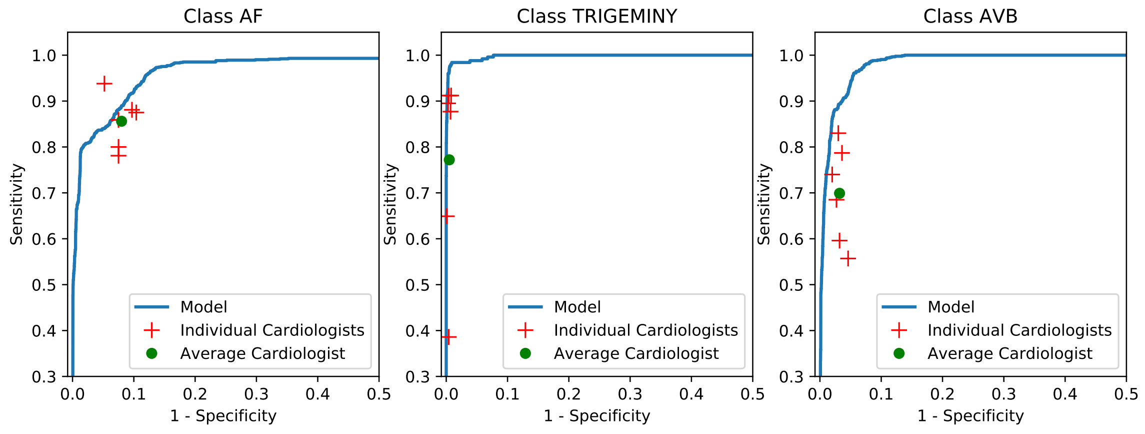


Figure 3.2: Receiver operating characteristic curves at the sequence level for atrial fibrillation and flutter (AF), trigeminy and atrioventricular block (AVB).

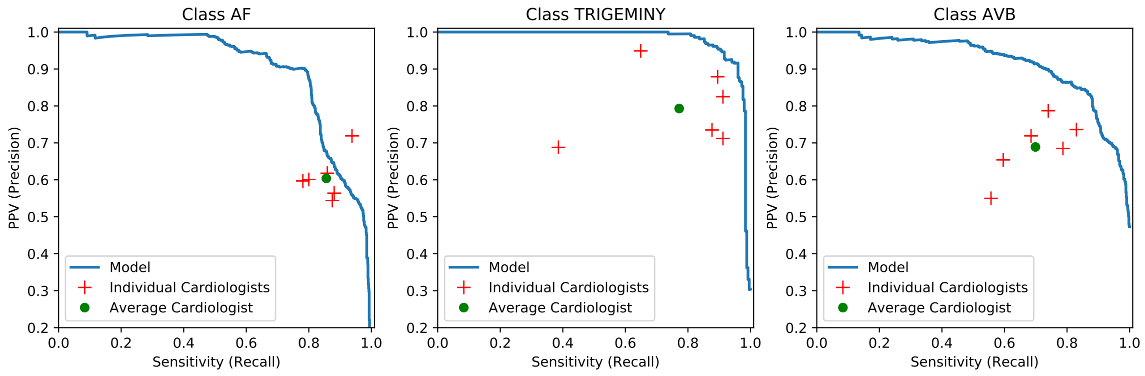
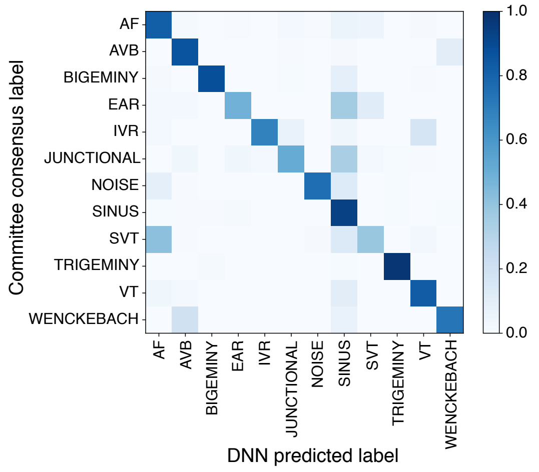


Figure 3.3: Precision-recall curves at the sequence level for atrial fibrillation and flutter (AF), trigeminy and atrioventricular block (AVB).

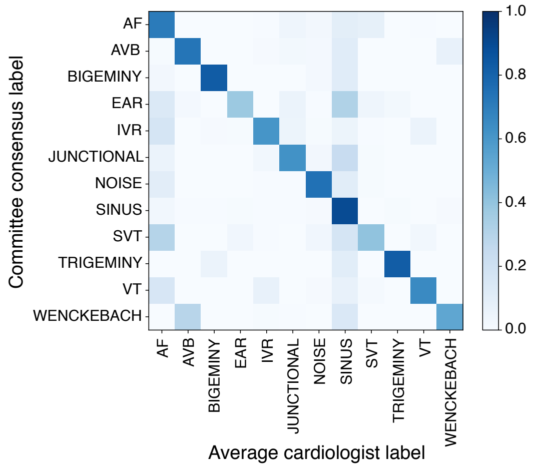
the other rhythm classes not shown here.

3.6 Analysis

The model outperforms the average cardiologist score on both the sequence and the set F1 metrics. Figure 3.4a shows a confusion matrix of the model predictions on the test set. Many arrhythmias are confused with the sinus rhythm. We expect that part of this is due to the sometimes ambiguous location of the exact onset and offset



(a) Model



(b) Individual Cardiologists

Figure 3.4: Confusion matrices for the model and individual cardiologist predictions with the committee consensus annotations as ground truth.

of the arrhythmia in the ECG record. Figure 3.4b shows a confusion matrix for the individual cardiologists on the test set. The high-level structure of the two confusion matrices is similar, suggesting that the model and cardiologists tend to make the same mistakes.

Often the mistakes made by the model are understandable. For example, confusing Wenckebach and AVB makes sense given that the two arrhythmias in general have very similar ECG morphologies. Similarly, Supraventricular Tachycardia (SVT) and Atrial Fibrillation / Flutter (AF) which is understandable given that they are all fast atrial arrhythmias. We also note that Idioventricular Rhythm (IVR) is sometimes mistaken as Ventricular Tachycardia (VT), which again makes sense given that the two only differ in heart-rate and are difficult to distinguish close to the 100 beats per minute delineation.

One of the most common confusions is between Ectopic Atrial Rhythm (EAR) and sinus rhythm. The main distinguishing criteria for this rhythm is an irregular P wave. This can be subtle to detect especially when the P wave has a small amplitude or when noise is present in the signal.

3.7 Conclusion

We develop a model which exceeds the cardiologist performance in detecting a wide range of heart arrhythmias from single-lead ECG records. Key to the performance of the model is a large annotated dataset and a very deep convolutional network which can map a sequence of ECG samples to a sequence of arrhythmia annotations.

On the clinical side, future work should investigate extending the set of arrhythmias and other forms of heart disease which can be automatically detected with high-accuracy from single or multiple lead ECG records. For example we do not detect Ventricular Flutter or Fibrillation. We also do not detect Left or Right Ventricular Hypertrophy, Myocardial Infarction or a number of other heart diseases which do not necessarily exhibit as arrhythmias. Some of these may be difficult or even impossible to detect on a single-lead ECG but can often be seen on a multiple-lead ECG.

Given that more than 300 million ECGs are recorded annually, high-accuracy diagnosis from ECG can save expert clinicians and cardiologists considerable time and decrease the number of misdiagnoses. Furthermore, we hope that this technology coupled with low-cost ECG devices enables more widespread use of the ECG as a diagnostic tool in places where access to a cardiologist is difficult.

Chapter 4

First-pass Speech Recognition

4.1 Introduction

Modern large vocabulary continuous speech recognition (LVCSR) systems are complex and difficult to modify. Much of this complexity stems from the paradigm of modeling words as sequences of sub-phonetic states with hidden Markov models (HMMs). HMM-based systems require carefully-designed training recipes to construct consecutively more complex HMM recognizers. The overall difficulty of building, understanding, and modifying HMM-based LVCSR systems has limited progress in speech recognition and isolated it from many advances in related fields.

Previous work demonstrated an HMM-free approach for training a speech recognizer [41]. The authors use a neural network to directly predict transcript characters given the audio of an utterance. This approach discards many of the assumptions present in modern HMM-based LVCSR systems in favor of treating speech recognition as a direct sequence transduction problem. The approach trains a neural network using the connectionist temporal classification (CTC) loss function, which amounts to maximizing the likelihood of an output sequence by efficiently summing over all possible input-output sequence alignments. Using CTC the authors were able to train a neural network to predict the character sequence of test utterances with a character error rate (CER) under 10% on the Wall Street Journal LVCSR corpus. While impressive in its own right, these results are not yet competitive with existing HMM-based systems in terms of word error rate (WER). Good word-level performance in speech recognition often depends heavily upon a language model to provide a prior probability over likely word sequences.

To integrate language model information during decoding, the probabilities from the CTC-trained neural network are used to rescore a lattice or n-best hypothesis list generated by a state-of-the-art HMM-based system [41]. This introduces a potentially confounding factor because an n-best list constrains the set of possible transcriptions significantly. Additionally, it results in an overall system which still relies on HMM speech recognition infrastructure to achieve the final results. In contrast, we present *first-pass* decoding results which use a neural network and language model to decode from scratch, rather than re-ranking an existing set of hypotheses.

We describe a decoding algorithm which directly integrates a language model with CTC-trained neural networks to search through the space of possible word sequences. Our first-pass decoding algorithm enables CTC-trained models to benefit from a language model without relying on an existing HMM-based system to generate a word lattice. This removes the lingering dependence on HMM-centric speech recognition toolkits and enables us to achieve fairly competitive WER results with only a neural network and n-gram language model.

Deep neural networks (DNNs) are the most widely used neural network architecture for speech recognition [55]. DNNs are a fairly generic architecture for classification and regression problems. In HMM-based LVCSR systems, DNNs act as acoustic models by predicting the HMM's hidden state given the acoustic input for a point in time. However, in such HMM-DNN systems the temporal reasoning about an output sequence takes place within the HMM rather than the neural network. CTC training of neural networks forces the network to model output sequence dependencies rather than reasoning about single time frames independently from others. To better handle such temporal dependencies previous work with CTC used long short term memory (LSTM) networks. The LSTM architecture was originally designed to prevent the vanishing gradient problem of sigmoidal DNNs or recurrent neural networks (RNNs) [57].

Our work uses vanilla RNNs instead of LSTMs as a neural network architecture. Vanilla RNNs are simpler overall, because there are only dense weight matrix connections between subsequent layers. This simpler architecture is more amenable to graphics processing unit (GPU) computing which can significantly reduce training times. Recent work shows that with rectifier nonlinearities DNNs can perform well in DNN-HMM systems without suffering from vanishing gradient problems during optimization [28, 141, 87]. This makes us hopeful that vanilla RNNs with rectifier nonlinearities may be able to perform comparably to LSTMs which are specially engineered to avoid vanishing gradients.

4.2 Model

We train neural networks using the CTC loss function to do maximum likelihood training of letter sequences given acoustic features as input. We consider a single utterance as a training example consisting of an acoustic feature matrix X and word transcription Y. The CTC objective function maximizes the log probability $\log p(Y \mid X)$.

4.2.1 Deep Neural Networks

With the loss function fixed we must next define how we compute $p(c \mid x_t)$, the predicted distribution over output characters c given the audio features x_t at time t. While many function approximators are possible for this task, we choose as our most basic model a DNN. A DNN computes the distribution $p(c \mid x_t)$ using a series of hidden layers followed by an output layer. Given an input vector x_t the first hidden layer activations are a vector computed as,

$$h^{(1)} = \sigma(W^{(1)} x_t + b^{(1)}).$$

The matrix $W^{(1)}$ and vector $b^{(1)}$ are the weight matrix and bias vector for the layer. The function $\sigma(\cdot)$ is a point-wise nonlinearity. We use rectifier nonlinearities and thus choose, $\sigma(z) = \max(z, 0)$.

DNNs can have arbitrarily many hidden layers. After the first hidden layer, the hidden activations $h^{(l)}$ for layer l are computed as,

$$h^{(l)} = \sigma(W^{(l)} h^{(l-1)} + b^{(l)}).$$

To obtain a proper distribution over the set of possible characters, the final layer of the network is a *softmax* output layer of the form,

$$p(c = c_k \mid x_t) = \frac{\exp(-(W_k^{(s)} h^{(L)} + b_k^{(s)}))}{\sum_j \exp(-(W_j^{(s)} h^{(L)} + b_j^{(s)}))},$$

where $W_k^{(s)}$ is the k'th row of the output weight matrix $W^{(s)}$, $b_k^{(s)}$ is a scalar bias term and L is the index of the last hidden layer.

We can compute a subgradient for all parameters of the DNN given a training example and thus utilize gradient-based optimization techniques. Note that this same DNN formulation is commonly used in DNN-HMM models to predict a distribution over senones instead of characters.

4.2.2 Recurrent Deep Neural Networks

A transcription W has many temporal dependencies which a DNN may not sufficiently capture. At each timestep t the DNN computes its output using only the input features x_t, ignoring previous hidden representations and output distributions. To enable better modeling of the temporal dependencies present in a problem, we use a RNN. We select one hidden layer l to have a temporally recurrent weight matrix $W^{(f)}$ and compute the layer's hidden activations as,

$$h_t^{(l)} = \sigma(W^{(l)} h_t^{(l-1)} + W^{(f)} h_{t-1}^{(l)} + b^{(l)}).$$

Note that we now make the distinction $h_t^{(l)}$ for the hidden activation vector of layer l at timestep t since it now depends upon the activation vector of layer l at time $t-1$.

When working with RNNs, we found it important to use a modified version of the rectifier nonlinearity. This modified function selects $\sigma(z) = \min(\max(z, 0), 20)$ which clips large activations to prevent divergence during network training. Setting the maximum allowed activation to 20 results in the clipped rectifier acting as a normal rectifier function in all but the most extreme cases.

Aside from these changes, computations for a RNN are the same as those in a DNN as described in 4.2.1. Like the DNN, we can compute a subgradient for a RNN using backpropagation through time. In our experiments we always compute the gradient completely through time rather than truncating to obtain an approximate subgradient.

4.2.3 Bi-Directional Recurrent Deep Neural Networks

While forward recurrent connections reflect the temporal nature of the audio input, a more powerful sequence transduction model is a bidirectional RNN (BRNN), which maintains state both forwards and backwards in time. Such a model can integrate information from the entire temporal extent of the input features when making each prediction. We extend the RNN to form a BRNN by again choosing a temporally recurrent layer l. The BRNN creates both a forward and backward intermediate hidden representation which we call $h_t^{(f)}$ and $h_t^{(b)}$ respectively.

We use the temporal weight matrices $W^{(f)}$ and $W^{(b)}$ to propagate $h_t^{(f)}$ forward in time and $h_t^{(b)}$ backward in time respectively. We update the forward and backward components via the equations,

$$\begin{aligned} h_t^{(f)} &= \sigma(W^{(j)T} h_t^{(j-1)} + W^{(f)T} h_{t-1}^{(f)} + b^{(j)}), \\ h_t^{(b)} &= \sigma(W^{(j)T} h_t^{(j-1)} + W^{(b)T} h_{t+1}^{(b)} + b^{(j)}). \end{aligned}$$

Note that the recurrent forward and backward hidden representations are computed entirely independently from each other. As with the RNN we use the modified non-linearity function $\sigma(z) = \min(\max(z, 0), 20)$. To obtain the final representation $h_t^{(l)}$ for the layer we sum the two temporally recurrent components,

$$h_t^{(l)} = h_t^{(f)} + h_t^{(b)}.$$

Aside from this change to the recurrent layer the BRNN computes its output using the same equations as the RNN. As for other models, we can compute a subgradient for the BRNN directly to perform gradient-based optimization.

4.3 Decoding

Assuming an input of length T, the output of the neural network will be $p(c \mid x_t)$ for $t = 1, \ldots, T$. Again, $p(c \mid x_t)$ is a distribution over possible characters in the alphabet Σ, which includes ϵ (the blank symbol), given audio input x_t. In order to recover a character string from the output of the neural network, as a first approximation, we take the $\arg\max$ at each time step. Let $S = (s_1, \ldots, s_T)$ be the character sequence where $s_t = \arg\max_{c \in \Sigma} p(c \mid x_t)$. The sequence S is mapped to a transcription by collapsing repeat characters and removing ϵ's. This gives a sequence which can be scored against the reference transcription using both CER and WER.

This first approximation lacks the ability to include the constraint of either a lexicon or a language model. We propose a generic algorithm which is capable of incorporating such constraints. Taking X to be the full acoustic input, we seek a transcription Y which maximizes the probability,

$$p_{\text{ctc}}(Y \mid X) p_{\text{lm}}(Y). \tag{4.1}$$

Here the overall probability of the transcription is modeled as the product of two factors: p_{ctc} given by the network and p_{lm} given by a language model prior. In practice the prior $p_{\text{lm}}(Y)$, when given by an n-gram language model, is too constraining and thus we down-weight it and include a word insertion penalty (or bonus) as

$$p_{\text{ctc}}(Y \mid X) p_{\text{lm}}(Y)^{\alpha} |Y|^{\beta}. \tag{4.2}$$

Alogrithm 1 attempts to find a transcript Y which maximizes equation 4.2.

The algorithm maintains two separate probabilities for each prefix, $p_b(\ell \mid x_{1:t})$ and $p_{nb}(\ell \mid x_{1:t})$. Respectively, these are the probability of the prefix ℓ ending in ϵ or not ending in ϵ given the first t time steps of the audio input X.

The sets A_{prev} and A_{next} maintain a list of active prefixes at the previous time step and proposed prefixes at the next time step respectively. Note that the size of A_{prev} is never larger than the beam width k. The overall probability of a prefix is the product of a word insertion term and the sum of the ϵ and non-ϵ ending probabilities,

$$p(\ell \mid x_{1:t}) = (p_b(\ell \mid x_{1:t}) + p_{nb}(\ell \mid x_{1:t})) |W(\ell)|^{\beta}, \tag{4.3}$$

where $W(\ell)$ is the set of words in the sequence ℓ. When taking the k most probable prefixes of A_{next}, we sort each prefix using the probability given by equation 4.3.

The variable ℓ_{end} is the last character in the label sequence ℓ. The function $W(\cdot)$, which converts ℓ into a string of words, segments the sequence ℓ at each space character and truncates any characters trailing the last space.

We incorporate a lexicon or language model constraint by including the probability

Algorithm 1 A beam search decoding algorithm for a CTC trained model. The algorithm also incorporates the constraint given by a dictionary or language model. The output is the most probable transcript, although this can easily be extended to return an n-best list.

```
$p_b(\emptyset \mid x_{1:0}) \leftarrow 1,\ p_{nb}(\emptyset \mid x_{1:0}) \leftarrow 0$
$A_{\text{prev}} \leftarrow \{\emptyset\}$
**for** $t = 1, \ldots, T$ **do**
    $A_{\text{next}} \leftarrow \{\}$
    **for** $\ell$ **in** $A_{\text{prev}}$ **do**
        **for** $c$ **in** $\Sigma$ **do**
            **if** $c = \epsilon$ **then**
                $p_b(\ell \mid x_{1:t}) \leftarrow p(\epsilon \mid x_t)(p_b(\ell \mid x_{1:t-1}) + p_{nb}(\ell \mid x_{1:t-1}))$
                add $\ell$ to $A_{\text{next}}$
            **else**
                $\ell^+ \leftarrow$ concatenate $\ell$ and $c$
                **if** $c = \ell_{\text{end}}$ **then**
                    $p_{nb}(\ell^+ \mid x_{1:t}) \leftarrow p(c \mid x_t)p_b(\ell \mid x_{1:t-1})$
                    $p_{nb}(\ell \mid x_{1:t}) \leftarrow p(c \mid x_t)p_b(\ell \mid x_{1:t-1})$
                **else if** $c =$ space **then**
                    $p_{nb}(\ell^+ \mid x_{1:t}) \leftarrow p(W(\ell^+) \mid W(\ell))^{\alpha} \cdot$
                        $p(c \mid x_t)(p_b(\ell \mid x_{1:t-1}) + p_{nb}(\ell \mid x_{1:t-1}))$
                **else**
                    $p_{nb}(\ell^+ \mid x_{1:t}) \leftarrow p(c \mid x_t)(p_b(\ell \mid x_{1:t-1}) + p_{nb}(\ell \mid x_{1:t-1}))$
                **end if**
                **if** $\ell^+$ **not in** $A_{\text{prev}}$ **then**
                    $p_b(\ell^+ \mid x_{1:t}) \leftarrow p(\epsilon \mid x_t)(p_b(\ell^+ \mid x_{1:t-1}) + p_{nb}(\ell^+ \mid x_{1:t-1}))$
                    $p_{nb}(\ell^+ \mid x_{1:t}) \leftarrow p(c \mid x_t)p_{nb}(\ell^+ \mid x_{1:t-1})$
                **end if**
                add $\ell^+$ to $A_{\text{next}}$
            **end if**
        **end for**
    **end for**
    $A_{\text{prev}} \leftarrow k$ most probable prefixes in $A_{\text{next}}$
**end for**
**return** 1 most probable prefix in $A_{\text{prev}}$
```

$p(W(\ell^+) \mid W(\ell))$ whenever the algorithm proposes appending a space character to ℓ. By setting $p(W(\ell^+) \mid W(\ell))$ to 1 if the last word of $W(\ell^+)$ is in the lexicon and 0 otherwise, the probability acts as a constraint forcing all character strings ℓ to consist of only words in the lexicon. Furthermore, $p(W(\ell^+) \mid W(\ell))$ can represent a n-gram language model by considering only the last $n-1$ words in $W(\ell)$.

4.4 Experiments

We evaluate our approach on the 81 hour Wall Street Journal (WSJ) news article dictation corpus (available in the LDC catalog as LDC94S13B and LDC93S6B). Our training set consists of 81 hours of speech from 37,318 utterances. The basic preparation of transforming the LDC-released corpora into training and test subsets follows the Kaldi speech recognition toolkit's s5 recipe [105]. However, we did not apply much of the text normalization used to prepare transcripts for training an HMM system. Instead we simply drop unnecessary transcript notes like lexical stress, keeping transcribed word fragments and acronym punctuation marks. We can safely discard much of this normalization because our approach does not rely on a lexicon or pronunciation dictionary, which cause problems especially for word fragments. Our language models are the standard models released with the WSJ corpus without lexical expansion. We used the 'dev93' evaluation subset as a development set and report final test set performance on the 'eval92' evaluation subset. Both subsets use the same 20k word vocabulary. The language model used for decoding is constrained to this same 20k word vocabulary.

The input audio was converted into log-Mel filterbank features with 23 frequency bins. A context window of +/- 10 frames were concatenated to form a final input vector of size 483. We did not perform additional feature preprocessing or feature-space speaker adaptation. Our output alphabet consists of 32 classes, namely the blank symbol "_", 26 letters, 3 punctuation marks (apostrophe, ., and -) as well as tokens for noise and space.

Model	CER	WER
No LM	10.0	35.8
Dictionary LM	8.5	24.4
Bigram LM	5.7	14.1

Table 4.1: Word error rate (WER) and character error rate (CER) results for a BRNN trained with the CTC loss function.

4.4.1 First-Pass Decoding with a Language Model

We trained a BRNN with 5 hidden layers, all with 1824 hidden units, for a total of 20.9M free parameters. The third hidden layer of the network has recurrent connections. Weights in the network are initialized from a uniform random distribution scaled by the weight matrix's input and output layer size [38]. We use the Nesterov accelerated gradient optimization algorithm [127] with initial learning rate 10^{-5}, and maximum momentum 0.95. After each full pass through the training set we divide the learning rate by 1.2 to ensure the overall learning rate decreases over time. We train the network for a total of 20 passes over the training set, which takes about 96 hours using our Python GPU implementation. For decoding with prefix search we use a beam size of 200 and cross-validate with a held-out set to find a good setting of the parameters α and β. Table 4.1 shows word and character error rates for multiple approaches to decoding with this trained BRNN.

Without any sort of language constraint WER is quite high, despite the fairly low CER. This is consistent with our observation that many mistakes at the character level occur when a word appears mostly correct but does not conform to the highly irregular orthography of English. Prefix-search decoding using the 20k word vocabulary as a prior over possible character sequences results in a substantial WER improvement, but changes the CER relatively little. Comparing the CERs of the no LM and dictionary LM approaches again demonstrates that without an LM the characters are mostly correct but are distributed across many words which increases WER. A large relative drop in both CER and WER occur when we decode with a bigram LM. Performance of the bigram LM model demonstrates that CTC-trained systems can attain competitive

Model	Parameters (M)	Train CER	Test CER
DNN	16.8	3.8	22.3
RNN	22.0	4.2	13.5
BRNN	20.9	2.8	10.7

Table 4.2: Train and test set CER results for a DNN without recurrence, an RNN, and a BRNN.

error rates without relying on a lattice or n-best list generated by an existing speech system.

4.4.2 The Effect of Recurrent Connections

Previous experiments with DNN-HMM systems found minimal benefits from recurrent connections in DNN acoustic models. It is natural to wonder whether recurrence, and especially bi-directional recurrence, is an essential aspect of our architecture. To evaluate the impact of recurrent connections we compare the train and test CER of the DNN, RNN, and BRNN models while roughly controlling for the total number of free parameters in the model. Table 4.2 shows the results for each type of architecture.

Both variants of recurrent models show substantial test set CER improvements over the non-recurrent DNN model. Note that we report performance for a DNN of only 16.8M total parameters which is smaller than the total number of parameters used in both the RNN and BRNN models. We found that larger DNNs performed worse on the test set, suggesting that DNNs may be more prone to over-fitting for this task. Although the BRNN has fewer parameters than the RNN it performs better on both the training and test sets. Again this suggests that the architecture itself drives improved performance rather than the total number of free parameters. Conversely, because the gap between bi-directional recurrence and single recurrence is small relative to a non-recurrent DNN, on-line speech recognition using a singly recurrent network may be feasible without overly damaging performance.

4.5 Conclusion

We presented a decoding algorithm which enables first-pass LVCSR with a language model for CTC-trained neural networks. This decoding approach removes the lingering dependence on HMM-based systems found in previous work. Furthermore, first-pass decoding demonstrates the capabilities of a CTC-trained system without the confounding factor of potential effects from pruning the search space via a provided lattice. While our results do not outperform the best HMM-based systems on the WSJ corpus, they demonstrate the promise of CTC-based speech recognition systems.

Our experiments with the BRNN further simplify the infrastructure needed to create CTC-based speech recognition systems. The BRNN is overall a less complex architecture than LSTMs and can relatively easily be made to run on GPUs since large matrix multiplications dominate the computation. However, our experiments suggest that recurrent connections are critical for good performance. Bi-directional recurrence helps beyond unidirectional recurrence but could be sacrificed in cases that require low-latency, online speech recognition. Taken together with previous work on CTC-based LVCSR, we believe there is an exciting path forward for high quality LVCSR without the complexity of HMM-based infrastructure.

Chapter 5

Deep Speech: Scaling Up End-to-end Speech Recognition

5.1 Introduction

Top speech recognition systems rely on sophisticated pipelines composed of multiple algorithms and hand-engineered processing stages. In this paper, we describe an end-to-end speech system, called “Deep Speech”, where deep learning supersedes these processing stages. Combined with a language model, this approach achieves higher performance than traditional methods on hard speech recognition tasks while also being much simpler. These results are made possible by training a large recurrent neural network (RNN) using multiple GPUs and thousands of hours of data. Because this system learns directly from data, we do not require specialized components for speaker adaptation or noise filtering. In fact, in settings where robustness to speaker variation and noise are critical, our system excels: Deep Speech outperforms previously published methods on the Switchboard Hub5’00 corpus, achieving 16.0 WER, and performs better than commercial systems in noisy speech recognition tests.

Traditional speech systems use many heavily engineered processing stages, including specialized input features, acoustic models, and Hidden Markov Models (HMMs). To improve these pipelines, domain experts must invest a great deal of effort tuning their features and models. The introduction of deep learning algorithms [81, 93, 55, 30] has improved speech system performance, usually by improving acoustic models. While this improvement has been significant, deep learning still plays only a limited role in traditional speech pipelines. As a result, to improve performance on a task such as recognizing speech in a noisy environment, one must laboriously engineer the rest of the system for robustness. In contrast, our system applies deep learning end-to-end using recurrent neural networks. We take advantage of the capacity provided by deep learning systems to learn from large datasets to improve our overall performance. Our model is trained end-to-end to produce transcriptions and thus, with sufficient data and computing power, can learn robustness to noise or speaker variation on its own.

Tapping the benefits of end-to-end deep learning, however, poses several challenges: (i) we must find innovative ways to build large, labeled training sets and (ii) we must be able to train networks that are large enough to effectively utilize all of this data. One challenge for handling labeled data in speech systems is finding the alignment of text transcripts with input speech. This problem has been addressed by Graves, Fernández, Gomez and Schmidhuber [40], thus enabling neural networks to easily consume unaligned, transcribed audio during training. Meanwhile, rapid training of large neural networks has been tackled by Coates et al. [26], demonstrating the speed advantages of multi-GPU computation. We aim to leverage these insights to fulfill the vision of a generic learning system, based on large speech datasets and scalable RNN training, that can surpass more complicated traditional methods. This vision is inspired partly by the work of Lee et. al. [81] who applied early unsupervised feature learning techniques to replace hand-built speech features.

We have chosen our RNN model specifically to map well to GPUs and we use a novel model partition scheme to improve parallelization. Additionally, we propose a process for assembling large quantities of labeled speech data exhibiting the distortions that our system should learn to handle. Using a combination of collected and synthesized data, our system learns robustness to realistic noise and speaker variation (including Lombard Effect [65]). Taken together, these ideas suffice to build an end-to-end speech system that is at once simpler than traditional pipelines yet also performs better on difficult speech tasks. Deep Speech achieves a word error rate of 16.0 on the full Switchboard Hub5'00 test set—the best published result. Further, on a new noisy speech recognition dataset of our own construction, our system achieves a word error rate of 19.1 where the best commercial systems achieve 30.5 error.

5.2 Related Work

Several parts of our work are inspired by previous results. Neural network acoustic models and other connectionist approaches were first introduced to speech pipelines in the early 1990s [11, 109, 33]. These systems, similar to DNN acoustic models [93, 55, 30], replace only one stage of the speech recognition pipeline. Mechanically, our system is similar to other efforts to build end-to-end speech systems from deep learning algorithms. For example, Graves et al. [40] have previously introduced the "Connectionist Temporal Classification" (CTC) loss function for scoring transcriptions produced by RNNs and, with LSTM networks, have previously applied this approach to speech [41]. We similarly adopt the CTC loss for part of our training procedure but use much simpler recurrent networks with rectified-linear activations [38, 87, 96]. Our recurrent network is similar to the bidirectional RNN used by Hannun et al. [49], but with multiple changes to enhance its scalability. By focusing on scalability, we have shown that these simpler networks can be effective even without the more complex LSTM machinery.

Our work is certainly not the first to exploit scalability to improve performance of DL algorithms. The value of scalability in deep learning is well-studied [27, 77] and the use of parallel processors (including GPUs) has been instrumental to recent large-scale DL results [129, 77]. Early ports of DL algorithms to GPUs revealed significant speed gains [107]. Researchers have also begun choosing designs that map well to GPU hardware to gain even more efficiency, including convolutional [74, 22, 113] and locally connected [26, 21] networks, especially when optimized libraries like cuDNN [14] and BLAS are available. Indeed, using high-performance computing infrastructure, it is possible today to train neural networks with more than 10 billion connections [26] using clusters of GPUs. These results inspired us to focus first on making scalable design choices to efficiently utilize many GPUs before trying to engineer the algorithms and models themselves.

With the potential to train large models, there is a need for large training sets as well. In other fields, such as computer vision, large labeled training sets have enabled significant leaps in performance as they are used to feed larger and larger DL systems [129, 74]. In speech recognition, however, such large training sets are less common, with typical benchmarks having training sets ranging from tens of hours (e.g. the Wall Street Journal corpus with 80 hours) to several hundreds of hours (e.g. Switchboard and Broadcast News). Larger benchmark datasets, such as the Fisher corpus [20] with 2000 hours of transcribed speech, are rare and only recently being studied. To fully utilize the expressive power of the recurrent networks available to us, we rely not only on large sets of labeled utterances, but also on synthesis techniques to generate novel examples. This approach is well known in computer vision [119, 80, 25] but we have found this especially convenient and effective for speech when done properly.

5.3 RNN Training Setup

The core of our system is a recurrent neural network (RNN) trained to ingest speech spectrograms and generate English text transcriptions. Let a single utterance X and label Y be sampled from a training set $\mathcal{X} = \{(X^{(1)}, Y^{(1)}), (X^{(2)}, Y^{(2)}), \ldots\}$. Each utterance, $X^{(i)}$, is a time-series of length $T^{(i)}$ where every time-slice is a vector of audio features, $x_t^{(i)}, t = 1, \ldots, T^{(i)}$. We use spectrograms as our features, so $x_{t,p}^{(i)}$ denotes the power of the p'th frequency bin in the audio frame at time

t. The goal of our RNN is to convert an input sequence X into a sequence of character probabilities for the transcription Y, with $\hat{y}_t = p(c_t \mid x)$, where $c_t \in \{\text{a,b,c}, \ldots, \text{z}, \textit{space}, \textit{apostrophe}, \textit{blank}\}$.

Our RNN model is composed of 5 layers of hidden units. For an input x, the hidden units at layer l are denoted $h^{(l)}$ with the convention that $h^{(0)}$ is the input. The first three layers are not recurrent. For the first layer, at each time t, the output depends on the spectrogram frame x_t along with a context of C frames on each side.[1] The remaining non-recurrent layers operate on independent data for each time step. Thus, for each time t, the first 3 layers are computed by:

$$h_t^{(l)} = g(W^{(l)} h_t^{(l-1)} + b^{(l)})$$

where $g(z) = \min\{\max\{0, z\}, 20\}$ is the clipped rectified-linear (ReLu) activation function and $W^{(l)}, b^{(l)}$ are the weight matrix and bias parameters for layer l.[2] The fourth layer is a bi-directional recurrent layer [120]. This layer includes two sets of hidden units: a set with forward recurrence, $h^{(f)}$, and a set with backward recurrence $h^{(b)}$:

$$\begin{aligned} h_t^{(f)} &= g(W^{(4)} h_t^{(3)} + W_r^{(f)} h_{t-1}^{(f)} + b^{(4)}) \\ h_t^{(b)} &= g(W^{(4)} h_t^{(3)} + W_r^{(b)} h_{t+1}^{(b)} + b^{(4)}) \end{aligned}$$

Note that $h^{(f)}$ must be computed sequentially from $t = 1$ to $t = T^{(i)}$ for the i'th utterance, while the units $h^{(b)}$ must be computed sequentially in reverse from $t = T^{(i)}$ to $t = 1$.

The fifth (non-recurrent) layer takes both the forward and backward units as inputs $h_t^{(5)} = g(W^{(5)} h_t^{(4)} + b^{(5)})$ where $h_t^{(4)} = h_t^{(f)} + h_t^{(b)}$. The output layer is a standard softmax function that yields the predicted character probabilities for each time slice t and character k in the alphabet.

[1] We typically use $C \in \{5, 7, 9\}$ for our experiments.

[2] The ReLu units are clipped in order to keep the activations in the recurrent layer from exploding; in practice the units rarely saturate at the upper bound.

Once we have computed a prediction for $p(c_t \mid x)$, we compute the CTC loss [40] to measure the error in prediction. During training, we can evaluate the gradient of the loss with respect to the network outputs. From this point, computing the gradient with respect to all of the model parameters is done via back-propagation through the rest of the network. We use Nesterov's Accelerated gradient method for training [127].[3]

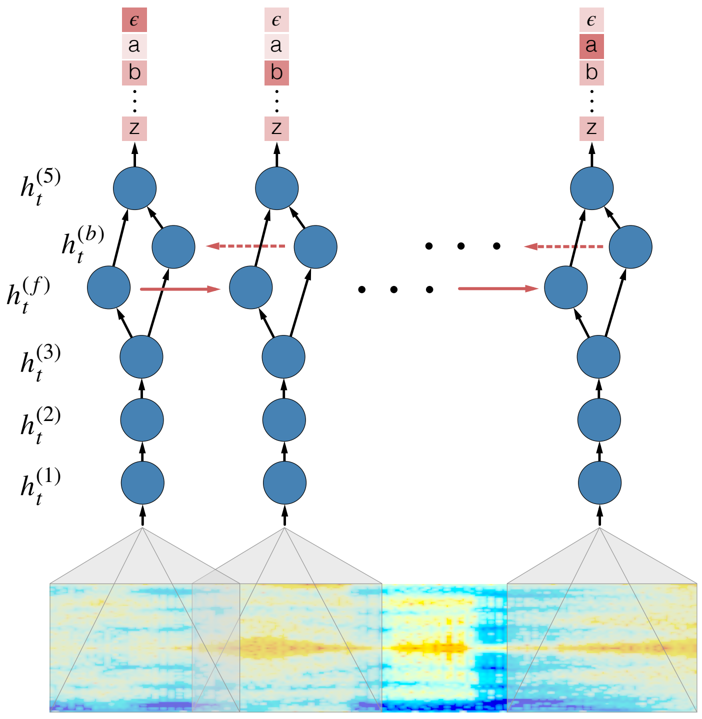


Figure 5.1: Structure of our RNN model and notation.

The complete RNN model is illustrated in Figure 5.1. Note that its structure is considerably simpler than related models from the literature [41]—we have limited ourselves to a single recurrent layer (which is the hardest to parallelize) and we do not use Long-Short-Term-Memory (LSTM) circuits. One disadvantage of LSTM cells is that they require computing and storing multiple gating neuron responses at each step. Since the forward and backward recurrences are sequential, this small additional cost can become a computational bottleneck. By using a homogeneous model we have

[3]We use momentum of 0.99 and anneal the learning rate by a constant factor, chosen to yield the fastest convergence, after each epoch through the data.

made the computation of the recurrent activations as efficient as possible: computing the ReLu outputs involves only a few highly optimized BLAS operations on the GPU and a single point-wise nonlinearity.

5.3.1 Regularization

While we have gone to significant lengths to expand our datasets (c.f. Section 5.5), the recurrent networks we use are still adept at fitting the training data. In order to reduce variance further, we use several techniques.

During training we apply a dropout [56] rate between 5% - 10%. We apply dropout in the feed-forward layers but not to the recurrent hidden activations.

A commonly employed technique in computer vision during network evaluation is to randomly jitter inputs by translations or reflections, feed each jittered version through the network, and vote or average the results [74]. Such jittering is not common in ASR, however we found it beneficial to translate the raw audio files by 5ms (half the filter bank step size) to the left and right, then forward propagate the recomputed features and average the output probabilities. At test time we also use an ensemble of several RNNs, averaging their outputs in the same way.

5.3.2 Language Model

When trained from large quantities of labeled speech data, the RNN model can learn to produce readable character-level transcriptions. Indeed for many of the transcriptions, the most likely character sequence predicted by the RNN is exactly correct without external language constraints. The errors made by the RNN in this case tend to be phonetically plausible renderings of English words—Table 5.1 shows some examples. Many of the errors occur on words that rarely or never appear in our training set. In practice, this is hard to avoid: training from enough speech data to *hear* all of the words or language constructions we might need to know is impractical. Therefore, we integrate our system with an n-gram language model since these models are easily trained from huge unlabeled text corpora. For comparison, while our speech datasets typically include up to 3 million utterances, the n-gram language model used for the experiments in Section 5.6.2 is trained from a corpus of 220 million phrases, supporting a vocabulary of 495,000 words.[4]

RNN output	Decoded Transcription
what is the weather like in bostin right now	what is the weather like in boston right now
prime miniter nerenr modi	prime minister narendra modi
arther n tickets for the game	are there any tickets for the game

Table 5.1: Examples of transcriptions directly from the RNN (left) with errors that are fixed by addition of a language model (right).

We perform a search to find the sequence of characters that is most probable according to both the RNN output and the language model (where the language model interprets the string of characters as words). Specifically, we aim to find a sequence Y that maximizes the combined objective:

$$Q(Y) = \log(p_{\text{ctc}}(Y \mid X)) + \alpha \log(p_{\text{lm}}(Y)) + \beta \text{ word\_count}(Y)$$

where α and β are tunable parameters (set by cross-validation) that control the trade-off between the RNN, the language model constraint and the length of the sentence. The term p_{lm} denotes the probability of the sequence Y according to the n-gram model. We maximize this objective using a highly optimized beam search algorithm, with a typical beam size in the range 1000-8000—similar to the approach described by Hannun et al. [49].

[4]We use the KenLM toolkit [53] to train the n-gram language models in our experiments.

5.4 Optimizations

As noted above, we have made several design decisions to make our networks amenable to high-speed execution (and thus fast training). For example, we have opted for homogeneous rectified-linear networks that are simple to implement and depend on just a few highly-optimized BLAS calls. When fully unrolled, our networks include almost 5 billion connections for a typical utterance and thus efficient computation is critical to make our experiments feasible. We use multi-GPU training [26, 74] to accelerate our experiments, but doing this effectively requires some additional work, as we explain.

5.4.1 Data parallelism

In order to process data efficiently, we use two levels of data parallelism. First, each GPU processes many examples in parallel. This is done in the usual way by concatenating many examples into a single matrix. For instance, rather than performing a single matrix-vector multiplication $W_r h_t$ in the recurrent layer, we prefer to do many in parallel by computing $W_r H_t$ where $H_t = [h_t^{(i)}, h_t^{(i+1)}, \ldots]$ (where $h_t^{(i)}$ corresponds to the i'th example $X^{(i)}$ at time t). The GPU is most efficient when H_t is relatively wide (e.g., 1000 examples or more) and thus we prefer to process as many examples on one GPU as possible (up to the limit of GPU memory).

When we wish to use larger minibatches than a single GPU can support on its own we use data parallelism across multiple GPUs, with each GPU processing a separate minibatch of examples and then combining its computed gradient with its peers during each iteration. We typically use $2\times$ or $4\times$ data parallelism across GPUs.

Data parallelism is not easily implemented, however, when utterances have different lengths since they cannot be combined into a single matrix multiplication. We resolve the problem by sorting our training examples by length and combining only similarly-sized utterances into minibatches, padding with silence when necessary so that all utterances in a batch have the same length. This solution is inspired by the ITPACK/ELLPACK sparse matrix format [68]; a similar solution was used by the Sutskever et al. [128] to accelerate RNNs for text.

5.4.2 Model parallelism

Data parallelism yields training speedups for modest multiples of the minibatch size (e.g., 2 to 4), but faces diminishing returns as batching more examples into a single gradient update fails to improve the training convergence rate. That is, processing $2\times$ as many examples on $2\times$ as many GPUs fails to yield a $2\times$ speedup in training. It is also inefficient to fix the total minibatch size but spread out the examples to $2\times$ as many GPUs: as the minibatch *within* each GPU shrinks, most operations become memory-bandwidth limited. To scale further, we parallelize by partitioning the model ("model parallelism" [26, 31]).

Our model is challenging to parallelize due to the sequential nature of the recurrent layers. Since the bidirectional layer is comprised of a forward computation and a backward computation that are independent, we can perform the two computations in parallel. Unfortunately, naively splitting the RNN to place $h^{(f)}$ and $h^{(b)}$ on separate GPUs commits us to significant data transfers when we go to compute $h^{(5)}$ (which depends on both $h^{(f)}$ and $h^{(b)}$). Thus, we have chosen a different partitioning of work that requires less communication for our models: we divide the model in half along the *time* dimension.

All layers except the recurrent layer can be trivially decomposed along the time dimension, with the first half of the time-series, from $t = 1$ to $t = T^{(i)}/2$, assigned to one GPU and the second half to another GPU. When computing the recurrent layer activations, the first GPU begins computing the forward activations $h^{(f)}$, while the second begins computing the backward activations $h^{(b)}$. At the mid-point ($t = T^{(i)}/2$), the two GPUs exchange the intermediate activations, $h^{(f)}_{T/2}$ and $h^{(b)}_{T/2}$ and swap roles. The first GPU then finishes the backward computation of $h^{(b)}$ and the second GPU

Dataset	Type	Hours	Speakers
WSJ	read	80	280
Switchboard	conversational	300	4000
Fisher	conversational	2000	23000
Baidu	read	5000	9600

Table 5.2: A summary of the datasets used to train Deep Speech. The WSJ, Switchboard and Fisher corpora are all published by the Linguistic Data Consortium.

finishes the forward computation of $h^{(f)}$.

5.4.3 Striding

We have worked to minimize the running time of the recurrent layers of our RNN, since these are the hardest to parallelize. As a final optimization, we shorten the recurrent layers by taking "steps" (or strides) of size 2 in the original input so that the unrolled RNN has half as many steps. This is similar to a convolutional network [78] with a step-size of 2 in the first layer. We use the cuDNN library [14] to implement this first layer of convolution efficiently.

5.5 Training Data

Large-scale deep learning systems require an abundance of labeled data. For our system we need many recorded utterances and corresponding English transcriptions, but there are few public datasets of sufficient scale. To train our largest models we have thus collected an extensive dataset consisting of 5000 hours of read speech from 9600 speakers. For comparison, we have summarized the labeled datasets available to us in Table 5.2.

5.5.1 Synthesis by superposition

To expand our potential training data even further we use data synthesis, which has been successfully applied in other contexts to amplify the effective number of training samples [119, 80, 25]. In our work, the goal is primarily to improve performance in noisy environments where existing systems break down. Capturing labeled data (e.g., read speech) from noisy environments is not practical, however, and thus we must find other ways to generate such data.

To a first order, audio signals are generated through a process of superposition of source signals. We can use this fact to synthesize noisy training data. For example, if we have a speech audio track $X^{(i)}$ and a "noise" audio track $\xi^{(i)}$, then we can form the "noisy speech" track $\hat{X}^{(i)} = X^{(i)} + \xi^{(i)}$ to simulate audio captured in a noisy environment. If necessary, we can add reverberations, echoes or other forms of damping to the power spectrum of $\xi^{(i)}$ or $X^{(i)}$ and then simply add them together to make fairly realistic audio scenes.

There are, however, some risks in this approach. For example, in order to take 1000 hours of clean speech and create 1000 hours of noisy speech, we will need unique noise tracks spanning roughly 1000 hours. We cannot settle for, say, 10 hours of repeating noise, since it may become possible for the recurrent network to memorize the noise track and "subtract" it out of the synthesized data. Thus, instead of using a single noise source $\xi^{(i)}$ with a length of 1000 hours, we use a large number of shorter clips (which are easier to collect from public video sources) and treat them as separate sources of noise before superimposing all of them: $\hat{X}^{(i)} = X^{(i)} + \xi_1^{(i)} + \xi_2^{(i)} + \ldots$.

When superimposing many signals collected from video clips, we can end up with "noise" sounds that are different from the kinds of noise recorded in real environments. To ensure a good match between our synthetic data and real data, we rejected any candidate noise clips where the average power in each frequency band differed significantly from the average power observed in real noisy recordings.

5.5.2 Capturing Lombard Effect

One challenging effect encountered by speech recognition systems in noisy environments is the “Lombard Effect” [65]: speakers actively change the pitch or inflections of their voice to overcome noise around them. This (involuntary) effect does not show up in recorded speech datasets since they are collected in quiet environments. To ensure that the effect is represented in our training data we induce the Lombard effect intentionally during data collection by playing loud background noise through headphones worn by a person as they record an utterance. The noise induces them to inflect their voice, thus allowing us to capture the Lombard effect in our training data.[5]

5.6 Experiments

We performed two sets of experiments to evaluate our system. In both cases we use the model described in Section 5.3 trained from a selection of the datasets in Table 5.2 to predict character-level transcriptions. The predicted probability vectors and language model are then fed into our decoder to yield a word-level transcription, which is compared with the ground truth transcription to yield the word error rate (WER).

5.6.1 Conversational speech: Switchboard Hub5’00 (full)

To compare our system to prior research we use an accepted but highly challenging test set, Hub5’00 (LDC2002S23). Some researchers split this set into “easy” (Switchboard) and “hard” (CallHome) instances, often reporting new results on the easier

[5]We have experimented with noise played through headphones as well as through computer speakers. Using headphones has the advantage that we obtain “clean” recordings without the background noise included and can add our own synthetic noise afterward.

portion alone. We use the full set, which is the most challenging case and report the overall word error rate.

We evaluate our system trained on only the 300 hour Switchboard conversational telephone speech dataset and trained on both Switchboard (SWB) and Fisher (FSH) [20], a 2000 hour corpus collected in a similar manner as Switchboard. Many researchers evaluate models trained only with 300 hours from Switchboard conversational telephone speech when testing on Hub5'00. In part this is because training on the full 2000 hour Fisher corpus is computationally difficult. Using the techniques mentioned in Section 5.4 our system is able perform a full pass over the 2300 hours of data in just a few hours.

Since the Switchboard and Fisher corpora are distributed at a sample rate of 8kHz, we compute spectrograms of 80 linearly spaced log filter banks and an energy term. The filter banks are computed over windows of 20ms strided by 10ms. We did not evaluate more sophisticated features such as the mel-scale log filter banks or the mel-frequency cepstral coefficients.

Speaker adaptation is critical to the success of current ASR systems [132, 114], particularly when trained on 300 hour Switchboard. For the models we test on Hub5'00, we apply a simple form of speaker adaptation by normalizing the spectral features on a per speaker basis. Other than this, we do not modify the input features in any way.

For decoding, we use a 4-gram language model with a 30,000 word vocabulary trained on the Fisher and Switchboard transcriptions. Again, hyperparameters for the decoding objective are chosen via cross-validation on a held-out development set.

The Deep Speech SWB model is a network of 5 hidden layers each with 2048 neurons trained on only 300 hour switchboard. The Deep Speech SWB + FSH model is an ensemble of 4 RNNs each with 5 hidden layers of 2304 neurons trained on the full 2300 hour combined corpus. All networks are trained on inputs of +/- 9 frames of context.

We report results in Table 5.3. The model from Vesely et al. (DNN-GMM sMBR) [132] uses a sequence based loss function on top of a DNN after using a typical hybrid DNN-HMM system to realign the training set. The performance of this model on the combined Hub5'00 test set is the best previously published result. When trained on the combined 2300 hours of data the Deep Speech system improves upon this baseline by 2.4 absolute WER and 13.0% relative. The model from Maas et al. (DNN-HMM FSH) [89] achieves 19.9 WER when trained on the Fisher 2000 hour corpus. That system was built using Kaldi [105], state-of-the-art open source speech recognition software. We include this result to demonstrate that Deep Speech, when trained on a comparable amount of data is competitive with the best existing ASR systems.

Model	SWB	CH	Full
Vesely et al. (GMM-HMM BMMI) [132]	18.6	33.0	25.8
Vesely et al. (DNN-HMM sMBR) [132]	12.6	24.1	18.4
Maas et al. (DNN-HMM SWB) [89]	14.6	26.3	20.5
Maas et al. (DNN-HMM FSH) [89]	16.0	23.7	19.9
Seide et al. (CD-DNN) [121]	16.1	n/a	n/a
Kingsbury et al. (DNN-HMM sMBR HF) [70]	13.3	n/a	n/a
Sainath et al. (CNN-HMM) [114]	11.5	n/a	n/a
Soltau et al. (MLP/CNN+I-Vector) [125]	**10.4**	n/a	n/a
Deep Speech SWB	20.0	31.8	25.9
Deep Speech SWB + FSH	12.6	**19.3**	**16.0**

Table 5.3: Published error rates (WER) on Switchboard dataset splits. The columns labeled "SWB" and "CH" are respectively the easy and hard subsets of Hub5'00.

5.6.2 Noisy speech

Few standards exist for testing noisy speech performance, so we constructed our own evaluation set of 100 noisy and 100 noise-free utterances from 10 speakers. The noise environments included a background radio or TV; washing dishes in a sink; a crowded cafeteria; a restaurant; and inside a car driving in the rain. The utterance text came primarily from web search queries and text messages, as well as news clippings, phone conversations, Internet comments, public speeches, and movie scripts. We did not have precise control over the signal-to-noise ratio (SNR) of the noisy samples, but we aimed for an SNR between 2 and 6 dB.

For the following experiments, we train our RNNs on all the datasets (more than 7000 hours) listed in Table 5.2. Since we train for 15 to 20 epochs with newly synthesized noise in each pass, our model learns from over 100,000 hours of novel data. We use an ensemble of 6 networks each with 5 hidden layers of 2560 neurons. No form of speaker adaptation is applied to the training or evaluation sets. We normalize training examples on a per utterance basis in order to make the total power of each example consistent. The features are 160 linearly spaced log filter banks computed over windows of 20ms strided by 10ms and an energy term. Audio files are resampled to 16kHz prior to the featurization. Finally, from each frequency bin we remove the global mean over the training set and divide by the global standard deviation, primarily so the inputs are well scaled during the early stages of training.

As described in Section 5.3.2, we use a 5-gram language model for the decoding. We train the language model on 220 million phrases of the Common Crawl[6], selected such that at least 95% of the characters of each phrase are in the alphabet. Only the most common 495,000 words are kept, the rest remapped to an `UNKNOWN` token.

We compared the Deep Speech system to several commercial speech systems: (1) wit.ai, (2) Google Speech API, (3) Bing Speech and (4) Apple Dictation.[7]

Our test is designed to benchmark performance in noisy environments. This situation creates challenges for evaluating the web speech APIs: these systems will give no result at all when the SNR is too low or in some cases when the utterance is too long. Therefore we restrict our comparison to the subset of utterances for which all systems returned a non-empty result.[8] The results of evaluating each system on our test files appear in Table 5.4.

[6]commoncrawl.org

[7]wit.ai and Google Speech each have HTTP-based APIs. To test Apple Dictation and Bing Speech, we used a kernel extension to loop audio output back to audio input in conjunction with the OS X Dictation service and the Windows 8 Bing speech recognition API.

[8]This leads to much higher accuracies than would be reported if we attributed 100% error in cases where an API failed to respond.

To evaluate the efficacy of the noise synthesis techniques described in Section 5.5.1, we trained two RNNs, one on 5000 hours of raw data and the other trained on the same 5000 hours plus noise. On the 100 clean utterances both models perform about the same, 9.2 WER and 9.0 WER for the clean trained model and the noise trained model respectively. However, on the 100 noisy utterances the noisy model achieves 22.6 WER over the clean model's 28.7 WER, a 6.1 absolute and 21.3% relative improvement.

System	Clean (94)	Noisy (82)	Combined (176)
Apple Dictation	14.24	43.76	26.73
Bing Speech	11.73	36.12	22.05
Google API	6.64	30.47	16.72
wit.ai	7.94	35.06	19.41
Deep Speech	**6.56**	**19.06**	**11.85**

Table 5.4: Results (WER) for 5 systems evaluated on the original audio. Scores are reported *only* for utterances with predictions given by all systems. The number in parentheses next to each dataset, e.g. Clean (94), is the number of utterances scored.

5.7 Conclusion

We have presented an end-to-end deep learning-based speech system capable of outperforming existing state-of-the-art recognition pipelines in two challenging scenarios: clear, conversational speech and speech in noisy environments. Our approach is enabled particularly by multi-GPU training and by data collection and synthesis strategies to build large training sets exhibiting the distortions our system must handle (such as background noise and Lombard effect). Combined, these solutions enable us to build a data-driven speech system that is at once better performing than existing methods while no longer relying on the complex processing stages that had stymied further progress. We believe this approach will continue to improve as we capitalize on increased computing power and dataset sizes in the future.

Chapter 6

Deep Speech 2: End-to-End Speech Recognition in English and Mandarin

6.1 Introduction

Decades worth of hand-engineered domain knowledge has gone into current state-of-the-art automatic speech recognition (ASR) pipelines. A simple but powerful alternative solution is to train such ASR models end-to-end, using deep learning to replace most modules with a single model [48]. We present the second generation of our speech system that exemplifies the major advantages of end-to-end learning. The Deep Speech 2 ASR pipeline approaches or exceeds the accuracy of Amazon Mechanical Turk human workers on several benchmarks, works in multiple languages with little modification, and is deployable in a production setting. It thus represents a significant step towards a single ASR system that addresses the entire range of speech recognition contexts handled by humans. Since our system is built on end-to-end deep learning, we can employ a spectrum of deep learning techniques: capturing large training sets, training larger models with high performance computing, and methodically exploring the space of neural network architectures. We show that through these techniques we are able to reduce error rates of our previous end-to-end system [48] in English by up to 43%, and can also recognize Mandarin speech with high accuracy.

One of the challenges of speech recognition is the wide range of variability in speech and acoustics. As a result, modern ASR pipelines are made up of numerous components including complex feature extraction, acoustic models, language and pronunciation models, speaker adaptation, etc. Building and tuning these individual components makes developing a new speech recognizer very hard, especially for a new language. Indeed, many parts do not generalize well across environments or languages and it is often necessary to support multiple application-specific systems in order to provide acceptable accuracy. This state of affairs is different from human speech recognition: people have the innate ability to learn any language during childhood, using general skills to learn language. After learning to read and write, most humans can transcribe speech with robustness to variation in environment, speaker accent and noise, without additional training for the transcription task. To meet the expectations of speech recognition users, we believe that a single engine must learn to be similarly competent; able to handle most applications with only minor modifications and able to learn new languages from scratch without dramatic changes. Our end-to-end system puts this goal within reach, allowing us to approach or exceed the performance of human workers on several tests in two very different languages: Mandarin and English.

Since Deep Speech 2 (DS2) is an end-to-end deep learning system, we can achieve performance gains by focusing on three crucial components: the model architecture, large labeled training datasets, and computational scale. This approach has also yielded great advances in other application areas such as computer vision and natural language. This paper details our contribution to these three areas for speech recognition, including an extensive investigation of model architectures and the effect of data and model size on recognition performance. In particular, we describe numerous experiments with neural networks trained with the Connectionist Temporal

Classification (CTC) loss function [40] to predict speech transcriptions from audio. We consider networks composed of many layers of recurrent connections, convolutional filters, and nonlinearities, as well as the impact of a specific instance of Batch Normalization [61] (BatchNorm) applied to RNNs. We not only find networks that produce much better predictions than those in previous work [48], but also find instances of recurrent models that can be deployed in a production setting with no significant loss in accuracy.

Beyond the search for better model architecture, deep learning systems benefit greatly from large quantities of training data. We detail our data capturing pipeline that has enabled us to create larger datasets than what is typically used to train speech recognition systems. Our English speech system is trained on 11,940 hours of speech, while the Mandarin system is trained on 9,400 hours. We use data synthesis to further augment the data during training.

Training on large quantities of data usually requires the use of larger models. Indeed, our models have many more parameters than those used in our previous system. Training a single model at these scales requires tens of exaFLOPs[1] that would require 3-6 weeks to execute on a single GPU. This makes model exploration a very time consuming exercise, so we have built a highly optimized training system that uses 8 or 16 GPUs to train one model. In contrast to previous large-scale training approaches that use parameter servers and asynchronous updates [31, 15], we use synchronous SGD, which is easier to debug while testing new ideas, and also converges faster for the same degree of data parallelism. To make the entire system efficient, we describe optimizations for a single GPU as well as improvements to scalability for multiple GPUs. We employ optimization techniques typically found in High Performance Computing to improve scalability. These optimizations include a fast implementation of the CTC loss function on the GPU, and a custom memory allocator. We also use carefully integrated compute nodes and a custom implementation of all-reduce to accelerate inter-GPU communication. Overall the system sustains approximately 50 teraFLOP/second when training on 16 GPUs. This amounts to 3 teraFLOP/second

[1]1 exaFLOP = 10^{18} FLoating-point OPerations.

per GPU which is about 50% of peak theoretical performance. This scalability and efficiency cuts training times down to 3 to 5 days, allowing us to iterate more quickly on our models and datasets.

We benchmark our system on several publicly available test sets and compare the results to our previous end-to-end system [48]. Our goal is to eventually reach human-level performance not only on specific benchmarks, where it is possible to improve through dataset-specific tuning, but on a range of benchmarks that reflects a diverse set of scenarios. To that end, we have also measured the performance of human workers on each benchmark for comparison. We find that our system outperforms humans in some commonly-studied benchmarks and has significantly closed the gap in much harder cases. In addition to public benchmarks, we show the performance of our Mandarin system on internal datasets that reflect real-world product scenarios.

Deep learning systems can be challenging to deploy at scale. Large neural networks are computationally expensive to evaluate for each user utterance, and some network architectures are more easily deployed than others. Through model exploration, we find high-accuracy, deployable network architectures, which we detail here. We also employ a batching scheme suitable for GPU hardware called Batch Dispatch that leads to an efficient, real-time implementation of our Mandarin engine on production servers. Our implementation achieves a 98th percentile compute latency of 67 milliseconds, while the server is loaded with 10 simultaneous audio streams.

6.2 Related Work

This work is inspired by previous work in both deep learning and speech recognition. Feed-forward neural network acoustic models were explored more than 20 years ago [11, 109, 33]. Recurrent neural networks and networks with convolution were also used in speech recognition around the same time [110, 133]. More recently DNNs have become a fixture in the ASR pipeline with almost all state of the art speech work containing some form of deep neural network [93, 55, 30, 29, 62, 121]. Convolutional networks have also been found beneficial for acoustic models [1, 114]. Recurrent neural networks, typically LSTMs, are just beginning to be deployed in state-of-the art recognizers [44, 116, 117] and work well together with convolutional layers for the feature extraction [115]. Models with both bidirectional [44] and unidirectional recurrence have been explored as well.

End-to-end speech recognition is an active area of research, showing compelling results when used to re-score the outputs of a DNN-HMM [41] and standalone [48]. Two methods are currently used to map variable length audio sequences directly to variable length transcriptions. The RNN encoder-decoder paradigm uses an encoder RNN to map the input to a fixed length vector and a decoder network to expand the fixed length vector into a sequence of output predictions [16, 128]. Adding an attentional mechanism to the decoder greatly improves performance of the system, particularly with long inputs or outputs [5]. In speech, the RNN encoder-decoder with attention performs well both in predicting phonemes [17] or graphemes [6, 12].

The other commonly used technique for mapping variable length audio input to variable length output is the CTC loss function [40] coupled with an RNN to model temporal information. The CTC-RNN model performs well in end-to-end speech recognition with grapheme outputs [41, 49, 48, 88]. The CTC-RNN model has also been shown to work well in predicting phonemes [92, 118], though a lexicon is still needed in this case. Furthermore it has been necessary to pre-train the CTC-RNN network with a DNN cross-entropy network that is fed frame-wise alignments from a GMM-HMM system [118]. In contrast, we train the CTC-RNN networks from scratch without the need of frame-wise alignments for pre-training.

Exploiting scale in deep learning has been central to the success of the field thus far [74, 77]. Training on a single GPU resulted in substantial performance gains [107], which were subsequently scaled linearly to two [74] or more GPUs [26]. We take advantage of work in increasing individual GPU efficiency for low-level deep learning primitives [14]. We build on the past work in using model-parallelism [26], data-parallelism [31] or a combination of the two [129, 48] to create a fast and highly scalable system for training deep RNNs in speech recognition.

Data has also been central to the success of end-to-end speech recognition, with over 7000 hours of labeled speech used in Deep Speech 1 (DS1) [48]. Data augmentation has been highly effective in improving the performance of deep learning in computer vision [80, 119, 25]. This has also been shown to improve speech systems [37, 48]. Techniques used for data augmentation in speech range from simple noise addition [48] to complex perturbations such as simulating changes to the vocal tract length and rate of speech of the speaker [63, 72].

Existing speech systems can also be used to bootstrap new data collection. In one approach, the authors use one speech engine to align and filter a thousand hours of read speech [102]. In another approach, a heavy-weight offline speech recognizer is used to generate transcriptions for tens of thousands of hours of speech [67]. This is then passed through a filter and used to re-train the recognizer, resulting in significant performance gains. Prior work has explored buildling accurate classifiers to estimate the confidence of a proposed transcription [59, 135]. These confidence measures can be used to construct high-quality filters for the bootstrapped dataset. We draw inspiration from these past approaches in bootstrapping larger datasets and data augmentation to increase the effective amount of labeled data for our system.

6.3 Model Architecture

A simple multi-layer model with a single recurrent layer cannot exploit thousands of hours of labelled speech. In order to learn from datasets this large, we increase the model capacity via depth. We explore architectures with up to 11 layers including many bidirectional recurrent layers and convolutional layers. These models have nearly 8 times the amount of computation per data example as the models in Deep Speech 1 making fast optimization and computation critical. In order to optimize these models successfully, we use Batch Normalization for RNNs and a novel optimization curriculum we call SortaGrad. We also exploit long *strides* between RNN inputs to reduce computation per example by a factor of 3. This is helpful for both training and evaluation, though requires some modifications in order to work well with CTC. Finally, though many of our research results make use of bidirectional recurrent layers, we find that excellent models exist using only unidirectional recurrent layers—a feature that makes such models much easier to deploy. Taken together these features allow us to tractably optimize deep RNNs and improve performance by more than 40% in both English and Mandarin error rates over the smaller baseline models.

6.3.1 Preliminaries

Figure 6.1 shows the architecture of the DS2 system which at its core is similar to the previous DS1 system [48]: a recurrent neural network (RNN) trained to ingest speech spectrograms and generate text transcriptions.

Let a single utterance X and label Y be sampled from a training set $\mathcal{X}$. We use a spectrogram of power normalized audio clips as the features to the system, so $x_{t,p}$ denotes the power of the p'th frequency bin in the audio frame at time t. The goal of the RNN is to convert an input sequence X into a final transcription Y.

The outputs of the network are the graphemes of each language. At each output time-step t, the RNN makes a prediction over characters, $p(\ell_t \mid x)$, where ℓ_t is either a character in the alphabet or the blank symbol. In English we have $\ell_t \in \{\text{a, b, c, } \ldots, \text{z}, \textit{space}, \textit{apostrophe}, \textit{blank}\}$. For the Mandarin system the network outputs simplified Chinese characters. We describe this in more detail in Section 6.3.9.

The RNN model is composed of several layers of hidden units. The architectures we experiment with consist of one or more convolutional layers, followed by one or more recurrent layers, followed by one or more fully connected layers.

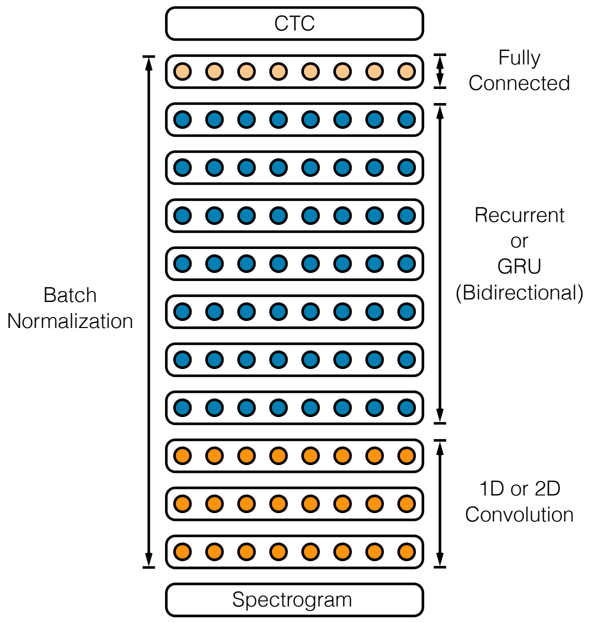


Figure 6.1: The architecture of the DS2 network used for both English and Mandarin speech.

The hidden representation at layer l is given by h^l with the convention that h^0 represents the input x. The bottom of the network is one or more convolutions over the time dimension of the input. For a context window of size c, the i-th activation at time-step t of the convolutional layer is given by

$$h^l_{t,i} = f(w^l_i \circ h^{l-1}_{t-c:t+c})$$

where $\circ$ denotes the element-wise product between the i-th filter and the context window of the previous layers activations, and f denotes a unary nonlinear function. We use the clipped rectified-linear (ReLU) function $f(x) = \min\{\max\{x, 0\}, 20\}$ as our nonlinearity. In some layers, usually the first, we sub-sample by striding the convolution by s frames. The goal is to shorten the number of time-steps for the recurrent layers above.

Following the convolutional layers are one or more bidirectional recurrent layers [120]. The forward in time $\overrightarrow{h}^l$ and backward in time $\overleftarrow{h}^l$ recurrent layer activations are computed as

$$\overrightarrow{h}_t^l = g(h_t^{l-1}, \overrightarrow{h}_{t-1}^l)$$
$$\overleftarrow{h}_t^l = g(h_t^{l-1}, \overleftarrow{h}_{t+1}^l)$$

The two sets of activations are summed to form the output activations for the layer $h^l = \overrightarrow{h}^l + \overleftarrow{h}^l$. The function $g(\cdot)$ can be the standard recurrent operation

$$\overrightarrow{h}_t^l = g(h_t^{l-1}, \overrightarrow{h}_{t-1}^l) = f(W^l h_t^{l-1} + \overrightarrow{U}^l \overrightarrow{h}_{t-1}^l + b^l) \tag{6.1}$$

where W^l is the input-hidden weight matrix, $\overrightarrow{U}^l$ is the recurrent weight matrix and b^l is a bias term. In this case the input-hidden weights are shared for both directions of the recurrence. The function $g(\cdot)$ can also represent more complex recurrence operations such as the Long Short-Term Memory (LSTM) units [57] and the gated recurrent units (GRU) [16].

After the bidirectional recurrent layers we apply one or more fully connected layers with

$$h_t^l = f(W^l h_t^{l-1} + b^l)$$

The output layer L is a softmax computing a probability distribution over characters.

The model is trained using the CTC loss function [40]. Given an input-output pair (X, Y) and the current parameters of the network θ, we compute the loss function $\mathcal{L}(X, Y; \theta)$ and its derivative with respect to the parameters of the network $\nabla_\theta \mathcal{L}(X, Y; \theta)$. This derivative is then used to update the network parameters through the backpropagation through time algorithm.

In the following subsections we describe the architectural and algorithmic improvements made relative to DS1 [48]. Unless otherwise stated these improvements are language agnostic. We report results on an English speaker held out development set which is an internal dataset containing 2048 utterances of primarily read speech. All models are trained on datasets described in Section 6.4. We report Word Error Rate (WER) for the English system and Character Error Rate (CER) for the Mandarin system. In both cases we integrate a language model in a beam search decoding step as described in Section 6.3.8.

6.3.2 Batch Normalization for Deep RNNs

To efficiently scale our model as we scale the training set, we increase the depth of the networks by adding more hidden layers, rather than making each layer larger. Previous work has examined doing so by increasing the number of consecutive bidirectional recurrent layers [44]. We explore Batch Normalization (BatchNorm) as a technique to accelerate training for such networks [61] since they often suffer from optimization issues.

Recent research has shown that BatchNorm improves the speed of convergence of recurrent nets, without showing any improvement in generalization performance [76]. In contrast, we demonstrate that when applied to very deep networks of simple RNNs on large data sets, batch normalization substantially improves final generalization error while greatly accelerating training.

In a typical feed-forward layer containing an affine transformation followed by a non-linearity $f(\cdot)$, we insert a BatchNorm transformation by applying $f(\mathcal{B}(Wh))$ instead of $f(Wh + b)$, where

$$\mathcal{B}(x) = \gamma \frac{x - \mathrm{E}[x]}{(\mathrm{Var}[x] + \epsilon)^{1/2}} + \beta.$$

The terms E and Var are the empirical mean and variance over a minibatch. The bias b of the layer is dropped since its effect is cancelled by mean removal. The learnable parameters γ and β allow the layer to scale and shift each hidden unit as desired. The constant ϵ is small and positive, and is included only for numerical stability. In our convolutional layers the mean and variance are estimated over all the temporal output units for a given convolutional filter on a minibatch. The BatchNorm transformation reduces *internal covariate shift* by insulating a given layer from potentially uninteresting changes in the mean and variance of the layer's input.

Architecture	Hidden Units	Train		Dev	
		Baseline	BatchNorm	Baseline	BatchNorm
1 RNN, 5 total	2400	10.55	11.99	13.55	14.40
3 RNN, 5 total	1880	9.55	8.29	11.61	10.56
5 RNN, 7 total	1510	8.59	7.61	10.77	9.78
7 RNN, 9 total	1280	8.76	7.68	10.83	9.52

Table 6.1: A comparison of WER on a training and development set for various depths of RNN, with and without BatchNorm. All networks have 38 million parameters. The architecture "M RNN, N total" means M consecutive bidirectional RNN layers with N total layers in the network.

We consider two methods of extending BatchNorm to bidirectional RNNs [76]. A natural extension is to insert a BatchNorm transformation immediately before every non-linearity. Equation 6.1 then becomes

$$\overrightarrow{h}^l_t = f(\mathcal{B}(W^l h^{l-1}_t + \overrightarrow{U}^l \overrightarrow{h}^l_{t-1})).$$

In this case the mean and variance statistics are accumulated over a single time-step of the minibatch. The sequential dependence between time-steps prevents averaging over all time-steps. We find that this technique does not lead to improvements in optimization. We also tried accumulating an average over successive time-steps, so later time-steps are normalized over all present and previous time-steps. This also proved ineffective and greatly complicated backpropagation.

We find that *sequence-wise* normalization [76] overcomes these issues. The recurrent computation is given by

$$\overrightarrow{h}^l_t = f(\mathcal{B}(W^l h^{l-1}_t) + \overrightarrow{U}^l \overrightarrow{h}^l_{t-1}).$$

For each hidden unit, we compute the mean and variance statistics over all items in the minibatch over the length of the sequence. Figure 6.2 shows that deep networks converge faster with sequence-wise normalization. Table 6.1 shows that the performance improvement from sequence-wise normalization increases with the depth of

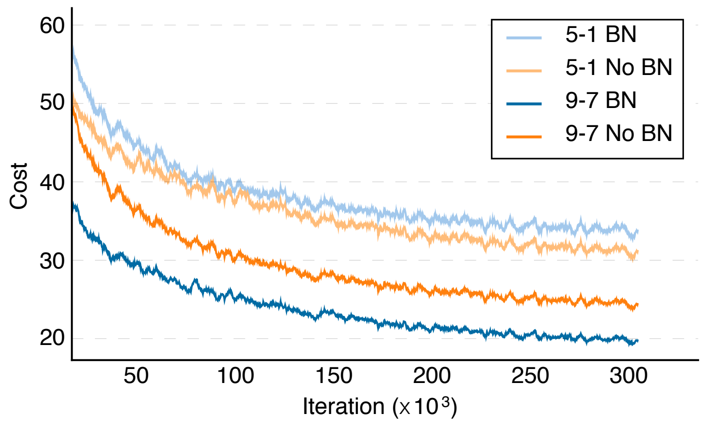


Figure 6.2: Training curves of two models trained with and without BatchNorm. We start the plot after the first epoch of training as the curve is more difficult to interpret due to the SortaGrad curriculum.

the network, with a 12% performance difference for the deepest network. When comparing depth, in order to control for model size we hold constant the total number of parameters and still see strong performance gains. We would expect to see even larger improvements from depth if we held constant the number of activations per layer and added layers. We also find that BatchNorm harms generalization error for the shallowest network just as it converges slower for shallower networks.

The BatchNorm approach works well in training, but is difficult to implement for a deployed ASR system, since it is often necessary to evaluate a single utterance in deployment rather than a batch. We find that normalizing each neuron to its mean and variance over just the sequence degrades performance. Instead, we store a running average of the mean and variance for the neuron collected during training, and use these for evaluation in deployment [61]. Using this technique, we can evaluate a single utterance at a time with better results than evaluating with a large batch.

6.3.3 SortaGrad

Training on examples of varying length pose some algorithmic challenges. One possible solution is truncating backpropagation through time [136], so that all examples

	Train		Dev	
	Baseline	BatchNorm	Baseline	BatchNorm
Not Sorted	10.71	8.04	11.96	9.78
Sorted	8.76	7.68	10.83	9.52

Table 6.2: Comparison of WER on a training and development set with and without SortaGrad, and with and without BatchNorm.

have the same sequence length during training [115]. However, this can inhibit the ability to learn longer term dependencies. Other works have found that presenting examples in order of difficulty can accelerate online learning [10, 140]. A common theme in many sequence learning problems including machine translation and speech recognition is that longer examples tend to be more challenging [16].

The CTC cost function that we use implicitly depends on the length of the utterance,

$$\mathcal{L}(X, Y; \theta) = -\log \sum_{\ell \in \text{Align}(X,Y)} \prod_{t}^{T} p_{\text{ctc}}(\ell_t \mid X; \theta). \tag{6.2}$$

where $\text{Align}(X, Y)$ is the set of all possible alignments of the characters of the transcription Y to frames of input X under the CTC operator. In equation 6.2, the inner term is a product over time-steps of the sequence, which shrinks with the length of the sequence since $p_{\text{ctc}}(\ell_t \mid X; \theta) < 1$. This motivates a curriculum learning strategy we title SortaGrad. SortaGrad uses the length of the utterance as a heuristic for difficulty, since long utterances have higher cost than short utterances.

In the first training epoch, we iterate through the training set in increasing order of the length of the longest utterance in the minibatch. After the first epoch, training reverts back to a random order over minibatches. Table 6.2 shows a comparison of training cost with and without SortaGrad on the 9 layer model with 7 recurrent layers. This effect is particularly pronounced for networks without BatchNorm, since they are numerically less stable. In some sense the two techniques substitute for one another, though we still find gains when applying SortaGrad and BatchNorm together. Even with BatchNorm we find that this curriculum improves numerical stability and sensitivity to small changes in training. Numerical instability can arise from different transcendental function implementations in the CPU and the GPU, especially when computing the CTC cost. This curriculum gives comparable results for both implementations.

We suspect that these benefits occur primarily because long utterances tend to have larger gradients, yet we use a fixed learning rate independent of utterance length. Furthermore, longer utterances are more likely to cause the internal state of the RNNs to explode at an early stage in training.

6.3.4 Comparison of simple RNNs and GRUs

The models we have shown so far are *simple* RNNs that have bidirectional recurrent layers with the recurrence for both the forward in time and backward in time directions modeled by Equation 6.1. Current research in speech and language processing has shown that having a more complex recurrence can allow the network to remember state over more time-steps while making them more computationally expensive to train [115, 12, 128, 6]. Two commonly used recurrent architectures are the Long Short-Term Memory (LSTM) units [57] and the Gated Recurrent Units (GRU) [16], though many other variations exist. A recent comprehensive study of thousands of variations of LSTM and GRU architectures showed that a GRU is comparable to an LSTM with a properly initialized forget gate bias, and their best variants are competitive with each other [64]. We decided to examine GRUs because experiments on smaller data sets showed the GRU and LSTM reach similar accuracy for the same number of parameters, but the GRUs were faster to train and less likely to diverge.

Architecture	Simple RNN	GRU
5 layers, 1 Recurrent	14.40	10.53
5 layers, 3 Recurrent	10.56	8.00
7 layers, 5 Recurrent	9.78	7.79
9 layers, 7 Recurrent	9.52	8.19

Table 6.3: Comparison of development set WER for networks with either simple RNN or GRU, for various depths.

The GRUs we use are computed by

$$\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{h}_t &= f(W_h x_t + r_t \circ U_h h_{t-1} + b_h) \\
h_t &= (1 - z_t) h_{t-1} + z_t \tilde{h}_t
\end{aligned}$$

where $\sigma(\cdot)$ is the sigmoid function, z and r represent the *update* and *reset* gates respectively, and we drop the layer superscripts for simplicity. We differ slightly from the standard GRU in that we multiply the hidden state h_{t-1} by U_h prior to scaling by the reset gate. This allows for all operations on h_{t-1} to be computed in a single matrix multiplication. The output nonlinearity $f(\cdot)$ is typically the hyperbolic tangent function *tanh*. However, we find similar performance for *tanh* and clipped-ReLU nonlinearities and choose to use the clipped-ReLU for simplicity and uniformity with the rest of the network.

Both GRU and simple RNN architectures benefit from batch normalization and show strong results with deep networks. However, Table 6.3 shows that for a fixed number of parameters, the GRU architectures achieve better WER for all network depths. This is clear evidence of the long term dependencies inherent in the speech recognition task present both within individual words and between words. As we discuss in Section 6.3.8, even simple RNNs are able to implicitly learn a language model due to the large amount of training data. Interestingly, the GRU networks with 5 or more recurrent layers do not significantly improve performance. We attribute this to the thinning from 1728 hidden units per layer for 1 recurrent layer to 768 hidden units per layer for 7 recurrent layers, to keep the total number of parameters constant.

The GRU networks outperform the simple RNNs in Table 6.3. However, in later results (Section 6.6) we find that as we scale up the model size, for a fixed computational budget the simple RNN networks perform slightly better. Given this, most of the remaining experiments use the simple RNN layers rather than the GRUs.

6.3.5 Frequency Convolutions

Temporal convolution is commonly used in speech recognition to efficiently model temporal translation invariance for variable length utterances. This type of convolution was first proposed for neural networks in speech more than 25 years ago [133]. Many neural network speech models have a first layer that processes input frames with some context window [29, 132]. This can be viewed as a temporal convolution with a stride of one.

Additionally, sub-sampling is essential to make recurrent neural networks computationally tractable with high sample-rate audio. The DS1 system accomplished this through the use of a spectrogram as input and temporal convolution in the first layer with a stride parameter to reduce the number of time-steps [48].

Convolutions in frequency and time domains, when applied to the spectral input features prior to any other processing, can slightly improve ASR performance [1, 114, 125]. Convolution in frequency attempts to model spectral variance due to speaker variability more concisely than what is possible with large fully connected networks. Since spectral ordering of features is removed by fully-connected and recurrent layers, frequency convolutions work better as the first layers of the network.

We experiment with adding between one and three layers of convolution. These are both in the time-and-frequency domain (2D invariance) and in the time-only domain

Layers	Channels	Filter dimension	Stride	Dev	Noisy Dev
1	1280	11	2	9.52	19.36
2	640, 640	5, 5	1, 2	9.67	19.21
3	512, 512, 512	5, 5, 5	1, 1, 2	9.20	20.22
1	32	41x11	2x2	8.94	16.22
2	32, 32	41x11, 21x11	2x2, 2x1	9.06	15.71
3	32, 32, 96	41x11, 21x11, 21x11	2x2, 2x1, 2x1	8.61	14.74

Table 6.4: Comparison of WER for various arrangements of convolutional layers. In all cases, the convolutions are followed by 7 recurrent layers and 1 fully connected layer. For 2D-invariant convolutions the first dimension is frequency and the second dimension is time.

(1D invariance). In all cases we use a *same* convolution, preserving the number of input features in both frequency and time. In some cases, we specify a stride across either dimension which reduces the size of the output. We do not explicitly control for the number of parameters, since convolutional layers add a small fraction of parameters to our networks. All networks shown in Table 6.4 have about 35 million parameters.

We report results on two datasets—a development set of 2048 utterances ("Regular Dev") and a much noisier dataset of 2048 utterances ("Noisy Dev") randomly sampled from the CHiME 2015 development datasets [7]. We find that multiple layers of 1D-invariant convolutions provides a very small benefit. The 2D-invariant convolutions improve results substantially on noisy data, while providing a small benefit on clean data. The change from one layer of 1D-invariant convolution to three layers of 2D-invariant convolution improves WER by 23.9% on the noisy development set.

6.3.6 Striding

In the convolutional layers, we apply a longer stride and wider context to speed up training as fewer time-steps are required to model a given utterance. Downsampling the input sound (through FFT and convolutional striding) reduces the number of time-steps and computation required in the following layers, but at the expense of reduced performance.

In our Mandarin models, we employ striding in the straightforward way. However, in English, striding can reduce accuracy simply because the output of our network requires at least one time-step per output character, and the number of characters in English speech per time-step is high enough to cause problems when striding[2]. To overcome this, we can enrich the English alphabet with symbols representing alternate labellings like whole words, syllables or non-overlapping n-grams. In practice, we use non-overlapping bi-graphemes or bigrams, since these are simple to construct, unlike syllables, and there are few of them compared to alternatives such as whole words. We transform unigram labels into bigram labels through a simple isomorphism.

Non-overlapping bigrams shorten the length of the output transcription and thus allow for a decrease in the length of the unrolled RNN. The sentence *the cat sat* with non-overlapping bigrams is segmented as $[th, e, space, ca, t, space, sa, t]$. Notice that for words with odd number of characters, the last character becomes an unigram and *space* is treated as an unigram as well. This isomorphism ensures that the same words are always composed of the same bigram and unigram tokens. The output set of bigrams consists of all bigrams that occur in the training set.

In Table 6.5 we show results for both the bigram and unigram systems for various levels of striding, with or without a language model. We observe that bigrams allow for larger strides without any sacrifice in in the word error rate. This allows us to reduce the number of time-steps of the unrolled RNN benefiting both computation and memory usage.

[2]Chinese characters are more similar to English syllables than English characters. This is reflected in our training data, where there are on average 14.1 characters/s in English, while only 3.3 characters/s in Mandarin. Conversely, the Shannon entropy per character as calculated from occurrence in the training set, is less in English due to the smaller character set—4.9 bits/char compared to 12.6 bits/char in Mandarin. This implies that spoken Mandarin has a lower temporal entropy density, $\sim$41 bits/s compared to $\sim$58 bits/s, and can thus more easily be temporally compressed without losing character information.

	Dev no LM		Dev LM	
Stride	Unigrams	Bigrams	Unigrams	Bigrams
2	14.93	14.56	9.52	9.66
3	15.01	15.60	9.65	10.06
4	18.86	14.84	11.92	9.93

Table 6.5: Comparison of WER with different amounts of striding for unigram and bigram outputs. The models are compared on a development set with and without the use of a 5-gram language model.

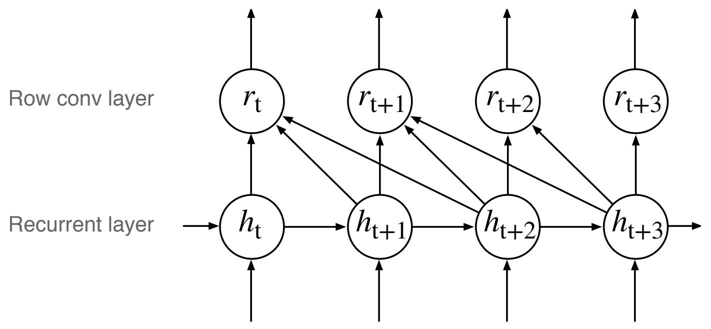


Figure 6.3: The row convolution architecture with two time-steps of future context.

6.3.7 Row Convolution and Unidirectional Models

Bidirectional RNN models are challenging to deploy in an online, low-latency setting, because they are built to operate on an entire sample, and so it is not possible to perform the transcription process as the utterance streams from the user. We have found an unidirectional architecture that performs as well as our bidirectional models. This allows us to use unidirectional, forward-only RNN layers in our deployment system.

To accomplish this, we employ a special layer that we call row convolution, shown in Figure 6.3. The intuition behind this layer is that we only need a small portion of future information to make an accurate prediction at the current time-step. Suppose at time-step t, we use a future context of τ steps. We now have a feature matrix $h_{t:t+\tau} = [h_t, h_{t+1}, ..., h_{t+\tau}]$ of size $d \times (\tau + 1)$. We define a parameter matrix W of the same size as $h_{t:t+\tau}$. The activations r_t for the new layer at time-step t are

$$r_{t,i} = \sum_{j=1}^{\tau+1} W_{i,j} h_{t+j-1,i}, \text{ for } 1 \leq i \leq d. \tag{6.3}$$

Since the convolution-like operation in Eq. 6.3 is row oriented for both W and $h_{t:t+\tau}$, we call this layer row convolution.

We place the row convolution layer above all recurrent layers. This has two advantages. First, this allows us to stream all computation below the row convolution layer on a finer granularity given little future context is needed. Second, this results in better CER compared to the best bidirectional model for Mandarin. We conjecture that the recurrent layers have learned good feature representations, so the row convolution layer simply gathers the appropriate information to feed to the classifier.

Results for a unidirectional Mandarin speech system with row convolution and a comparison to a bidirectional model are given in Section 6.7 on deployment.

6.3.8 Language Model

We train our RNN Models over millions of unique utterances, which enables the network to learn a powerful implicit language model. Our best models are quite adept at spelling, without any external language constraints. Further, in our development datasets we find many cases where our models can implicitly disambiguate homophones—for example, "he expects the Japanese agent *to* sell it for *two* hundred seventy five thousand dollars". Nevertheless, the labeled training data is small compared to the size of unlabeled text corpora that are available. Thus we find that WER improves when we supplement our system with a language model trained from external text.

We use an n-gram language model since they scale well to large amounts of unlabeled text [48]. For English, our language model is a Kneser-Ney smoothed 5-gram model

Language	Architecture	Dev no LM	Dev LM
English	5-layer, 1 RNN	27.79	14.39
English	9-layer, 7 RNN	14.93	9.52
Mandarin	5-layer, 1 RNN	9.80	7.13
Mandarin	9-layer, 7 RNN	7.55	5.81

Table 6.6: Comparison of WER for English and CER for Mandarin with and without a language model.

with pruning that is trained using the KenLM toolkit [53] on cleaned text from the Common Crawl Repository[3]. The vocabulary is the most frequently used 400,000 words from 250 million lines of text, which produces a language model with about 850 million n-grams. For Mandarin, the language model is a Kneser-Ney smoothed character level 5-gram model with pruning that is trained on an internal text corpus of 8 billion lines of text. This produces a language model with about 2 billion n-grams.

During inference we search for the transcription Y that maximizes $Q(Y)$ shown in Equation 6.4. This is a linear combination of log probabilities from the CTC trained network and language model, along with a word insertion term [48].

$$Q(Y) = \log(p_{\text{ctc}}(Y \mid X)) + \alpha \log(p_{\text{lm}}(Y)) + \beta \, \text{word\_count}(Y) \tag{6.4}$$

The weight α controls the relative contributions of the language model and the CTC network. The weight β encourages more words in the transcription. These parameters are tuned on a development set. We use a beam search to find the optimal transcription [49].

Table 6.6 shows that an external language model helps both English and Mandarin speech systems. The relative improvement given by the language model drops from 48% to 36% in English and 27% to 23% in Mandarin, as we go from a model with 5 layers and 1 recurrent layer to a model with 9 layers and 7 recurrent layers. We

[3]`http://commoncrawl.org`

hypothesize that the network builds a stronger implicit language model with more recurrent layers.

The relative performance improvement from a language model is higher in English than in Mandarin. We attribute this to the fact that a Chinese character represents a larger block of information than an English character. For example, if we output directly to syllables or words in English, the model would make fewer spelling mistakes and the language model would likely help less.

6.3.9 Adaptation to Mandarin

The techniques that we have described so far can be used to build an end-to-end Mandarin speech recognition system that outputs Chinese characters directly. This precludes the need to construct a pronunciation model, which is often a fairly involved component for porting speech systems to other languages [123]. Direct output to characters also precludes the need to explicitly model language specific pronunciation features. For example we do not need to model Mandarin tones explicitly, as some speech systems must do [123, 98].

The only architectural changes we make to our networks are due to the characteristics of the Chinese character set. Firstly, the output layer of the network outputs about 6000 characters, which includes the Roman alphabet, since hybrid Chinese-English transcripts are common. We incur an out of vocabulary error at evaluation time if a character is not contained in this set. This is not a major concern, as our test set has only 0.74% out of vocab characters.

We use a character level language model in Mandarin as words are not usually segmented in text. The word insertion term of Equation 6.4 becomes a character insertion term. In addition, we find that the performance of the beam search during decoding levels off at a smaller beam size. This allows us to use a beam size of 200 with a negligible degradation in CER. In Section 6.6.2, we show that our Mandarin speech models show roughly the same improvements to architectural changes as our English

Dataset	Speech Type	Hours
WSJ	read	80
Switchboard	conversational	300
Fisher	conversational	2000
LibriSpeech	read	960
Baidu	read	5000
Baidu	mixed	3600
Total		11940

Table 6.7: Summary of the datasets used to train DS2 in English. The WSJ, Switchboard and Fisher corpora are all published by the Linguistic Data Consortium. The LibriSpeech dataset is available free on-line. The other datasets are internal Baidu corpora.

speech models.

6.4 Data

Large-scale deep learning systems require an abundance of labeled training data. We have collected an extensive training dataset for both English and Mandarin speech models, in addition to augmenting our training with publicly available datasets. In English we use 11,940 hours of labeled speech data containing 8 million utterances summarized in Table 6.7. For the Mandarin system we use 9,400 hours of labeled audio containing 11 million utterances. The Mandarin speech data consists of internal Baidu corpora, representing a mix of read speech and spontaneous speech, in both standard Mandarin and accented Mandarin.

6.4.1 Dataset Construction

Some of the internal English (3,600 hours) and Mandarin (1,400 hours) datasets were created from raw data captured as long audio clips with noisy transcriptions. The length of these clips ranged from several minutes to more than hour, making it impractical to unroll them in time in the RNN during training. To solve this problem, we developed an alignment, segmentation and filtering pipeline that can generate a training set with shorter utterances and few erroneous transcriptions.

The first step in the pipeline is to use an existing bidirectional RNN model trained with CTC to align the transcription to the frames of audio. For a given audio-transcript pair, (X, Y), we find the alignment that maximizes

$$\ell^* = \operatorname*{arg\,max}_{\ell \in \text{Align}(X,Y)} \prod_t^T p_{\text{ctc}}(\ell_t \mid X; \theta).$$

This is essentially a Viterbi alignment found using a RNN model trained with CTC. Since Equation 6.2 integrates over the alignment, the CTC loss function is never explicitly asked to produce an accurate alignment. In principle, CTC could choose to emit all the characters of the transcription after some fixed delay and this can happen with unidirectional RNNs [118]. However, we found that CTC produces an accurate alignment when trained with a bidirectional RNN.

Following the alignment is a segmentation step that splices the audio and the corresponding aligned transcription whenever it encounters a long series of consecutive *blank* labels occurs, since this usually denotes a stretch of silence. By tuning the number of consecutive *blank*s, we can tune the length of the utterances generated. For the English speech data, we also require a *space* token to be within the stretch of *blank*s in order to segment only on word boundaries. We tune the segmentation to generate utterances that are on average 7 seconds long.

The final step in the pipeline removes erroneous examples that arise from a failed alignment. We crowd source the ground truth transcriptions for several thousand examples. The word level edit distance between the ground truth and the aligned transcription is used to produce a *good* or *bad* label. The threshold for the word level edit distance is chosen such that the resulting WER of the *good* portion of the development set is less than 5%. Initially we thresholded the raw CTC cost (log-likelihood) generated from a pretrained model. However, we find that training a linear classifier to accurately predict bad examples given several different features yields better results. We add the following additional features: the CTC cost normalized by the sequence length, the CTC cost normalized by the transcript length, the ratio of the sequence length to the transcript length, the number of words in the transcription and the number of characters in the transcription. For the English dataset, we find that the filtering pipeline reduces the WER from 17% to 5% while retaining more than 50% of the examples.

6.4.2 Data Augmentation

We augment our training data by adding noise to increase the effective size of our training data and to improve our robustness to noisy speech [48]. Although the training data contains some intrinsic noise, we can increase the quantity and variety of noise through augmentation. Too much noise augmentation tends to make optimization difficult and can lead to worse results, and too little noise augmentation makes the system less robust to low signal-to-noise speech. We find that a good balance is to add noise to 40% of the utterances that are chosen at random. The noise source consists of several thousand hours of randomly selected audio clips combined to produce hundreds of hours of noise.

6.4.3 Scaling Data

Our English and Mandarin corpora are substantially larger than those commonly reported in speech recognition literature. In Table 6.8, we show the effect of increasing the amount of labeled training data on WER. This is done by randomly sampling the full dataset before training. For each dataset, the model was trained for up to 20 epochs though usually early-stopped based on the error on a held out development set. We note that the WER decreases with a power law for both the regular and noisy development sets. The WER decreases by $\sim$40% relative for each factor of 10 increase in training set size. We also observe a consistent gap in WER ($\sim$60%

Fraction of Data	Hours	Regular Dev	Noisy Dev
1%	120	29.23	50.97
10%	1200	13.80	22.99
20%	2400	11.65	20.41
50%	6000	9.51	15.90
100%	12000	8.46	13.59

Table 6.8: Comparison of English WER for Regular and Noisy development sets on increasing training dataset size. The architecture is a 9-layer model with 2 layers of 2D-invariant convolution and 7 recurrent layers with 68M parameters.

relative) between the regular and noisy datasets, implying that more data benefits both cases equally.

This implies that a speech system will continue to improve with more labeled training data. We hypothesize that equally as important as increasing raw number of hours is increasing the number of speech *contexts* that are captured in the dataset. A context can be any property that makes speech unique including different speakers, background noise, environment, and microphone hardware. While we do not have the labels needed to validate this claim, we suspect that measuring WER as a function of speakers in the dataset would lead to much larger relative gains than simple random sampling.

6.5 System Optimizations

Our networks have tens of millions of parameters, and the training algorithm takes tens of single-precision exaFLOPs to converge. Since our ability to evaluate hypotheses about our data and models depends on the ability to train models quickly, we built a highly optimized training system. This system has two main components—a deep learning library written in C++ , along with a high-performance linear algebra library written in both CUDA and C++ . Our optimized software, running on dense compute nodes with 8 Titan X GPUs per node, allows us to sustain 24 single-precision teraFLOP/second when training a single model on one node. This is 45% of the theoretical peak computational throughput of each node. We also can scale to multiple nodes, as outlined in the next subsection.

6.5.1 Scalability and Data-Parallelism

We use the standard technique of data-parallelism to train on multiple GPUs using synchronous SGD. Our most common configuration uses a minibatch of 512 on 8 GPUs. Our training pipeline binds one process to each GPU. These processes then exchange gradient matrices during the backpropagation by using all-reduce, which exchanges a matrix between multiple processes and sums the result so that at the end, each process has a copy of the sum of all matrices from all processes.

We find synchronous SGD useful because it is reproducible and deterministic. We have found that the appearance of non-determinism in our system often signals a serious bug, and so having reproducibility as a goal has greatly facilitates debugging. In contrast, asynchronous methods such as asynchronous SGD with parameter servers as found in Dean et al. [31] typically do not provide reproducibility and are therefore more difficult to debug. Synchronous SGD is simple to understand and implement. It scales well as we add multiple nodes to the training process.

Figure 6.4 shows that time taken to train one epoch halves as we double the number of GPUs that we train on, thus achieving near-linear weak scaling. We keep the minibatch per GPU constant at 64 during this experiment, effectively doubling the minibatch as we double the number of GPUs. Although we have the ability to scale to large minibatches, we typically use either 8 or 16 GPUs during training with a minibatch of 512 or 1024, in order to converge to the best result.

Since all-reduce is critical to the scalability of our training, we wrote our own implementation of the ring algorithm [104, 130] for higher performance and better stability. Our implementation avoids extraneous copies between CPU and GPU, and is fundamental to our scalability. We configure OpenMPI with the *smcuda* transport that

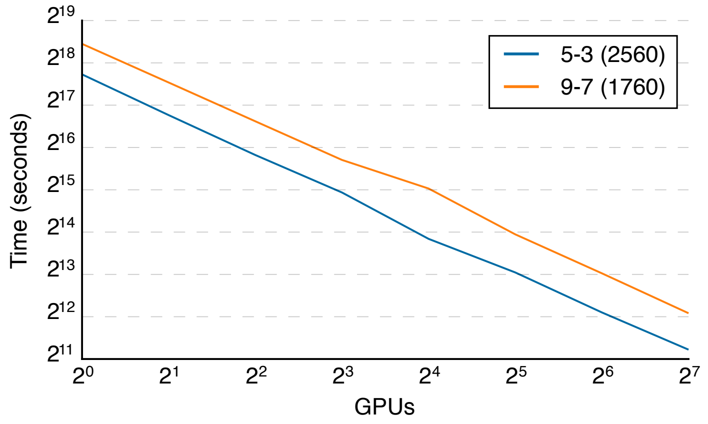


Figure 6.4: Scaling comparison of two networks—a 5 layer model with 3 recurrent layers containing 2560 hidden units in each layer and a 9 layer model with 7 recurrent layers containing 1760 hidden units in each layer. The times shown are to train 1 epoch.

can send and receive buffers residing in the memory of two different GPUs by using GPUDirect. When two GPUs are in the same PCI root complex, this avoids any unnecessary copies to CPU memory. This also takes advantage of tree-structured interconnects by running multiple segments of the ring concurrently between neighboring devices. We built our implementation using MPI send and receive, along with CUDA kernels for the element-wise operations.

Table 6.9 compares the performance of our all-reduce implementation with that provided by OpenMPI version 1.8.5. We report the time spent in all-reduce for a full training run that ran for one epoch on our English dataset using a 5 layer, 3 recurrent layer architecture with 2560 hidden units for all layers. In this table, we use a minibatch of 64 per GPU, expanding the algorithmic minibatch as we scale to more GPUs. We see that our implementation is considerably faster than OpenMPI's when the communication is within a node (8 GPUs or less). As we increase the number of GPUs and increase the amount of inter-node communication, the gap shrinks, although our implementation is still 2-4X faster.

All of our training runs use either 8 or 16 GPUs, and in this regime, our all-reduce implementation results in 2.5× faster training for the full training run, compared to using OpenMPI directly. Optimizing all-reduce has thus resulted in important

GPU	OpenMPI all-reduce	Our all-reduce	Performance Gain
4	55359.1	2587.4	21.4
8	48881.6	2470.9	19.8
16	21562.6	1393.7	15.5
32	8191.8	1339.6	6.1
64	1395.2	611.0	2.3
128	1602.1	422.6	3.8

Table 6.9: Comparison of two different all-reduce implementations. All times are in seconds. Performance gain is the ratio of OpenMPI all-reduce time to our all-reduce time.

productivity benefits for our experiments, and has made our simple synchronous SGD approach scalable.

6.5.2 GPU implementation of CTC loss function

Calculating the CTC loss function is more complicated than performing forward and back propagation on our RNN architectures. Originally, we transferred activations from the GPUs to the CPU, where we calculated the loss function using an OpenMP parallelized implementation of CTC. However, this implementation limited our scalability rather significantly, for two reasons. Firstly, it became computationally more significant as we improved efficiency and scalability of the RNN itself. Secondly, transferring large activation matrices between CPU and GPU required us to spend interconnect bandwidth for CTC, rather than on transferring gradient matrices to allow us to scale using data parallelism to more processors.

To overcome this, we wrote a GPU implementation of the CTC loss function. Our parallel implementation relies on a slight refactoring to simplify the dependences in the CTC calculation, as well as the use of optimized parallel sort implementations from ModernGPU [9].

Table 6.10 compares the performance of two CTC implementations. The GPU implementation saves us 95 minutes per epoch in English, and 25 minutes in Mandarin.

Language	Architecture	CPU CTC Time	GPU CTC Time	Speedup
English	5-layer, 3 RNN	5888.12	203.56	28.9
Mandarin	5-layer, 3 RNN	1688.01	135.05	12.5

Table 6.10: Comparison of time spent in seconds in computing the CTC loss function and gradient in one epoch for two different implementations. Speedup is the ratio of CPU CTC time to GPU CTC time.

This reduces overall training time by 10-20%, which is also an important productivity benefit for our experiments.

6.5.3 Memory allocation

Our system makes frequent use of dynamic memory allocations to GPU and CPU memory, mainly to store activation data for variable length utterances, and for intermediate results. Individual allocations can be very large; over 1 GB for the longest utterances. For these very large allocations we found that CUDA's memory allocator and even `std::malloc` introduced significant overhead into our application—over a 2x slowdown from using `std::malloc` in some cases. This is because both `cudaMalloc` and `std::malloc` forward very large allocations to the operating system or GPU driver to update the system page tables. This is a good optimization for systems running multiple applications, all sharing memory resources, but editing page tables is pure overhead for our system where nodes are dedicated entirely to running a single model. To get around this limitation, we wrote our own memory allocator for both CPU and GPU allocations. Our implementation follows the approach of the last level shared allocator in jemalloc: all allocations are carved out of contiguous memory blocks using the buddy algorithm [71]. To avoid fragmentation, we preallocate all of GPU memory at the start of training and subdivide individual allocations from this block. Similarly, we set the CPU memory block size that we forward to `mmap` to be substantially larger than `std::malloc`, at 12GB.

Most of the memory required for training deep recurrent networks is used to store activations through each layer for use by back propagation, not to store the parameters of the network. For example, storing the weights for a 70M parameter network with 9 layers requires approximately 280 MB of memory, but storing the activations for a batch of 64, seven-second utterances requires 1.5 GB of memory. TitanX GPUs include 12GB of GDDR5 RAM, and sometimes very deep networks can exceed the GPU memory capacity when processing long utterances. This can happen unpredictably, especially when the distribution of utterance lengths includes outliers, and it is desirable to avoid a catastrophic failure when this occurs. When a requested memory allocation exceeds available GPU memory, we allocate page-locked GPU-memory-mapped CPU memory using `cudaMallocHost` instead. This memory can be accessed directly by the GPU by forwarding individual memory transactions over PCIe at reduced bandwidth, and it allows a model to continue to make progress even after encountering an outlier.

The combination of fast memory allocation with a fallback mechanism that allows us to slightly overflow available GPU memory in exceptional cases makes the system significantly simpler, more robust, and more efficient.

6.6 Results

To better assess the real-world applicability of our speech system, we evaluate on a wide range of test sets. We use several publicly available benchmarks and several test sets collected internally. Together these test sets represent a wide range of challenging speech environments including low signal-to-noise ratios (noisy and far-field), accented, read, spontaneous and conversational speech.

All models are trained for 20 epochs on either the full English dataset, described in Table 6.7, or the full Mandarin dataset described in Section 6.4. We use stochastic gradient descent with Nesterov momentum [127] along with a minibatch of 512 utterances. If the norm of the gradient exceeds a threshold of 400, it is rescaled to 400 [103]. The model which performs the best on a held-out development set during training is chosen for evaluation. The learning rate is chosen from $[1 \times 10^{-4}, 6 \times 10^{-4}]$ to yield fastest convergence and annealed by a constant factor of 1.2 after each epoch. We use a momentum of 0.99 for all models.

The language models used are those described in Section 6.3.8. The decoding parameters from Equation 6.4 are tuned on a held-out development set. We use a beam size of 500 for the English decoder and a beam size of 200 for the Mandarin decoder.

6.6.1 English

The best DS2 model has 11 layers with 3 layers of 2D convolution, 7 bidirectional recurrent layers, a fully-connected output layer along with Batch Normalization. The first layer outputs to bigrams with a temporal stride of 3. By comparison the DS1 model has 5 layers with a single bidirectional recurrent layer and it outputs to unigrams with a temporal stride of 2 in the first layer. We report results on several test sets for both the DS2 and DS1 model. We do not tune or adapt either model to any of the speech conditions in the test sets. Language model decoding parameters are set once on a held-out development set.

To put the performance of our system in context, we benchmark most of our results against human workers, since speech recognition is an audio perception and language understanding problem that humans excel at. We obtain a measure of human level performance by paying workers from Amazon Mechanical Turk to hand-transcribe all of our test sets. Two workers transcribe the same audio clip, that is typically about 5 seconds long, and we use the better of the two transcriptions for the final WER calculation. They are free to listen to the audio clip as many times as they like. These workers are mostly based in the United States, and on average spend about 27 seconds per transcription. The hand-transcribed results are compared to the existing ground truth to produce a WER. While the existing ground truth transcriptions do have some label error, this is rarely more than 1%. This implies that disagreement between the ground truth transcripts and the human level transcripts is a good heuristic for human

Model size	Model type	Regular Dev	Noisy Dev
18×10^6	GRU	10.59	21.38
38×10^6	GRU	9.06	17.07
70×10^6	GRU	8.54	15.98
70×10^6	RNN	8.44	15.09
100×10^6	GRU	7.78	14.17
100×10^6	RNN	7.73	13.06

Table 6.11: Comparing the effect of model size on the WER of the English speech system on both the regular and noisy development sets. We vary the number of hidden units in all but the convolutional layers. The GRU model has 3 layers of bidirectional GRUs with 1 layer of 2D-invariant convolution. The RNN model has 7 layers of bidirectional recurrence with 3 layers of 2D-invariant convolution. Both models output bigrams with a temporal stride of 3.

level performance.

Model Size

Our English speech training set is substantially larger than the size of commonly used speech datasets. Furthermore, the data is augmented with noise synthesis. To get the best generalization error, we expect that the model size must increase to fully exploit the patterns in the data. In Section 6.3.2 we explored the effect of model depth while fixing the number of parameters. In contrast, here we show the effect of varying model size on the performance of the speech system. We only vary the size of each layer, while keeping the depth and other architectural parameters constant. We evaluate the models on the same Regular and Noisy development sets that we use in Section 6.3.5.

The models in Table 6.11 differ from those in Table 6.3 in that we increase the the stride to 3 and output to bigrams. Because we increase the model size to as many as 100 million parameters, we find that an increase in stride is necessary for fast computation and memory constraints. However, in this regime we note that the performance advantage of the GRU networks appears to diminish over the simple

Test set	DS1	DS2
Baidu Test	24.01	13.59

Table 6.12: Comparison of DS1 and DS2 WER on an internal test set of 3,300 examples.

RNN. In fact, for the 100 million parameter networks the simple RNN performs better than the GRU network and is faster to train despite the 2 extra layers of convolution.

Table 6.11 shows that the performance of the system improves consistently up to 100 million parameters. All further English DS2 results are reported with this same 100 million parameter RNN model since it achieves the lowest generalization errors.

Table 6.12 shows that the 100 million parameter RNN model (DS2) gives a 43.4% relative improvement over the 5-layer model with 1 recurrent layer (DS1) on an internal Baidu dataset of 3,300 utterances that contains a wide variety of speech including challenging accents, low signal-to-noise ratios from far-field or background noise, spontaneous and conversational speech.

Read Speech

Read speech with high signal-to-noise ratio is arguably the easiest large vocabulary for a continuous speech recognition task. We benchmark our system on two test sets from the Wall Street Journal (WSJ) corpus of read news articles. These are available in the LDC catalog as LDC94S13B and LDC93S6B. We also take advantage of the recently developed LibriSpeech corpus constructed using audio books from the LibriVox project [102].

Table 6.13 shows that the DS2 system outperforms humans in 3 out of the 4 test sets and is competitive on the fourth. Given this result, we suspect that there is little room for a generic speech system to further improve on clean read speech without further domain adaptation.

Read Speech			
Test set	DS1	DS2	Human
WSJ eval'92	4.94	3.60	5.03
WSJ eval'93	6.94	4.98	8.08
LibriSpeech test-clean	7.89	5.33	5.83
LibriSpeech test-other	21.74	13.25	12.69

Table 6.13: Comparison of WER for two speech systems and human level performance on read speech.

Accented Speech

Our source for accented speech is the publicly available VoxForge[4] dataset, which has clean speech read from speakers with many different accents. We group these accents into four categories. The American-Canadian and Indian groups are self-explanatory. The Commonwealth accent denotes speakers with British, Irish, South African, Australian and New Zealand accents. The European group contains speakers with accents from countries in Europe that do not have English as a first language. We construct a test set from the VoxForge data with 1024 examples from each accent group for a total of 4096 examples.

Performance on these test sets is to some extent a measure of the breadth and quality of our training data. Table 6.14 shows that our performance improved on all the accents when we include more accented training data and use an architecture that can effectively train on that data. However human level performance is still notably better than that of DS2 for all but the Indian accent.

Noisy Speech

We test our performance on noisy speech using the publicly available test sets from the recently completed third CHiME challenge [7]. This dataset has 1320 utterances

[4]`http://www.voxforge.org`

Accented Speech			
Test set	DS1	DS2	Human
VoxForge American-Canadian	15.01	7.55	4.85
VoxForge Commonwealth	28.46	13.56	8.15
VoxForge European	31.20	17.55	12.76
VoxForge Indian	45.35	22.44	22.15

Table 6.14: Comparing WER of the DS1 system to the DS2 system on accented speech.

Noisy Speech			
Test set	DS1	DS2	Human
CHiME eval clean	6.30	3.34	3.46
CHiME eval real	67.94	21.79	11.84
CHiME eval sim	80.27	45.05	31.33

Table 6.15: Comparison of DS1 and DS2 system on noisy speech. "CHiME eval clean" is a noise-free baseline. The "CHiME eval real" dataset is collected in real noisy environments and the "CHiME eval sim" dataset has similar noise synthetically added to clean speech.

from the WSJ test set read in various noisy environments, including a bus, a cafe, a street and a pedestrian area. The CHiME set also includes 1320 utterances with simulated noise from the same environments as well as the control set containing the same utterances delivered by the same speakers in a noise-free environment. Differences between results on the control set and the noisy sets provide a measure of the network's ability to handle a variety of real and synthetic noise conditions. The CHiME audio has 6 channels and using all of them can provide substantial performance improvements [139]. We use a *single* channel for all our results, since multi-channel audio is not pervasive on most devices. Table 6.15 shows that DS2 substantially improves upon DS1, however DS2 is worse than human level performance on noisy data. The relative gap between DS2 and human level performance is larger when the data comes from a real noisy environment instead of synthetically adding noise to clean speech.

6.6.2 Mandarin

In Table 6.16 we compare several architectures trained on the Mandarin Chinese speech, on a development set of 2000 utterances as well as a test set of 1882 examples of noisy speech. This development set was also used to tune the decoding parameters We see that the deepest model with 2D-invariant convolution and BatchNorm outperforms the shallow RNN by 48% relative, thus continuing the trend that we saw with the English system—multiple layers of bidirectional recurrence improves performance substantially.

Architecture	Dev	Test
5-layer, 1 RNN	7.13	15.41
5-layer, 3 RNN	6.49	11.85
5-layer, 3 RNN + BatchNorm	6.22	9.39
9-layer, 7 RNN + BatchNorm + 2D conv	5.81	7.93

Table 6.16: The effect of the DS2 architecture on Mandarin WER. The development and test sets are Baidu internal corpora.

We find that our best Mandarin Chinese speech system transcribes short voice-query like utterances better than a typical Mandarin Chinese speaker. To benchmark against humans we ran a test with 100 randomly selected utterances and had a group of 5 humans label all of them together. The group of humans had an error rate of 4.0% as compared to the speech systems performance of 3.7%. We also compared a single human transcriber to the speech system on 250 randomly selected utterances. In this case the speech system performs much better: 9.7% for the human compared to 5.7% for the speech model.

6.7 Deployment

Real-world applications usually require a speech system to transcribe in real time or with relatively low latency. The system used in Section 6.6.1 is not well-designed for this task, for several reasons. First, since the RNN has several bidirectional layers, transcribing the first part of an utterance requires the entire utterance to be presented to the RNN. Second, since we use a wide beam when decoding with a language model, beam search can be expensive, particularly in Mandarin where the number of possible next characters is very large (around 6000). Third, as described in Section 6.3, we normalize power across an entire utterance, which again requires the entire utterance to be available in advance.

We solve the power normalization problem by using some statistics from our training set to perform an adaptive normalization of speech inputs during online transcription. We can solve the other problems by modifying our network and decoding procedure to produce a model that performs almost as well while having much lower latency. We focus on our Mandarin system since some aspects of that system are more challenging to deploy (e.g. the large character set), but the same techniques could also be applied in English.

In this section, latency refers to the computational latency of our speech system as measured from the end of an utterance until the transcription is produced. This latency does not include data transmission over the internet, and does not measure latency from the beginning of an utterance until the first transcription is produced. We focus on latency from end of utterance to transcription because it is important to applications using speech recognition.

6.7.1 Batch Dispatch

In order to deploy our relatively large deep neural networks at low latency, we have paid special attention to efficiency during deployment. Most internet applications process requests individually as they arrive in the data center. This makes for a straightforward implementation where each request can be managed by one thread. However, processing requests individually is inefficient computationally, for two main reasons. Firstly, when processing requests individually, the processor must load all

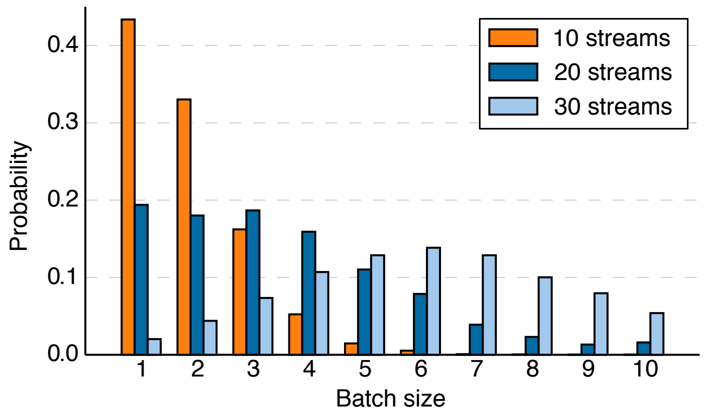


Figure 6.5: Probability that a request is processed in a batch of given size

the weights of the network for each request. This lowers the arithmetic intensity of the workload, and tends to make the computation memory bandwidth bound, as it is difficult to effectively use on-chip caches when requests are presented individually. Secondly, the amount of parallelism that can be exploited to classify one request is limited, making it difficult to exploit SIMD or multi-core parallelism. RNNs are especially challenging to deploy because evaluating RNNs sample by sample relies on sequential matrix vector multiplications, which are bandwidth bound and difficult to parallelize.

To overcome these issues, we built a batching scheduler called Batch Dispatch that assembles streams of data from user requests into batches before performing forward propagation on these batches. In this case, there is a tradeoff between increased batch size, and consequently improved efficiency, and increased latency. The more we buffer user requests to assemble a large batch, the longer users must wait for their results. This places constraints on the amount of batching we can perform.

We use an eager batching scheme that processes each batch as soon as the previous batch is completed, regardless of how much work is ready by that point. This scheduling algorithm has proved to be the best at reducing end-user latency, despite the fact that it is less efficient computationally, since it does not attempt to maximize batch size.

Figure 6.5 shows the probability that a request is processed in a batch of given size for our production system running on a single NVIDIA Quadro K1200 GPU, with 10-30 concurrent user requests. As expected, batching works best when the server is heavily loaded: as load increases, the distribution shifts to favor processing requests in larger batches. However, even with a light load of only 10 concurrent user requests, our system performs more than half the work in batches with at least 2 samples.

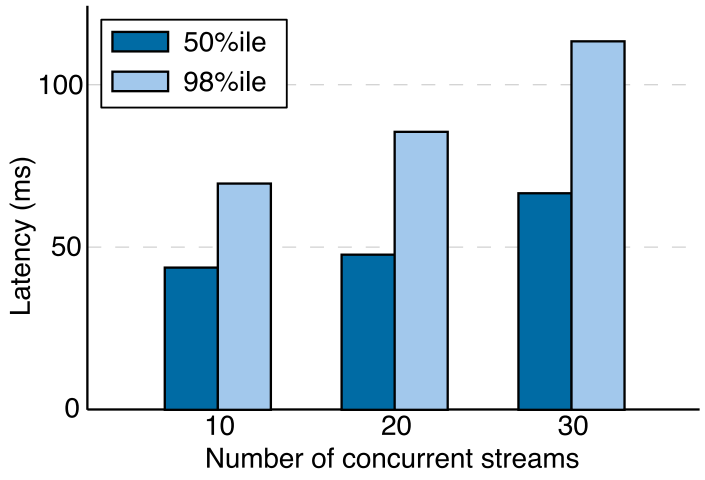


Figure 6.6: Median and 98th percentile latencies as a function of server load

We see in Figure 6.6, that our system achieves a median latency of 44 ms, and a 98 percentile latency of 70 ms when loaded with 10 concurrent streams. As the load increases on the server, the batching scheduler shifts work to more efficient batches, which keeps latency low. This shows that Batch Dispatch makes it possible to deploy these large models at high throughput and low latency.

6.7.2 Deployment Optimized Matrix Multiply Kernels

We have found that deploying our models using half-precision (16-bit) floating-point arithmetic does not measurably change recognition accuracy. Because deployment does not require any updates to the network weights, it is far less sensitive to numerical precision than training. Using half-precision arithmetic saves memory space

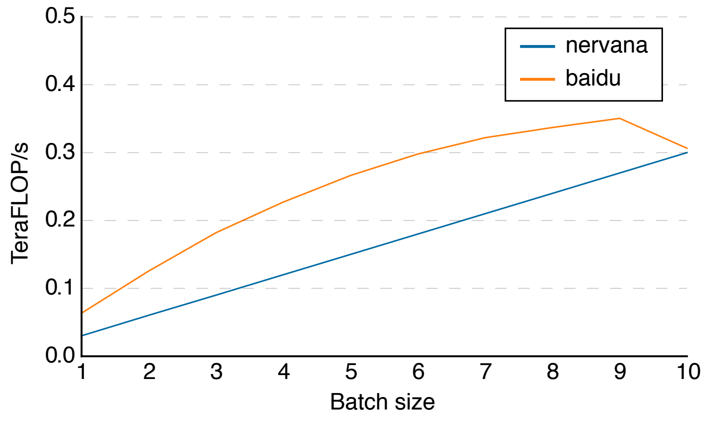


Figure 6.7: Comparison of kernels that compute $Ax = b$ where A is a matrix with dimension 2560×2560, and x is a matrix with dimension $2560 \times$ Batch size, where Batch size $\in [1, 10]$. All matrices are in half-precision format.

and bandwidth, which is especially useful for deployment, since RNN evaluation is dominated by the cost of caching and streaming the weight matrices.

The batch size during deployment is much smaller than in training. We found that standard BLAS libraries are inefficient at this batch size. To overcome this, we wrote our own half-precision matrix-matrix multiply kernel. For 10 simultaneous streams over 90 percent of batches are for $N \leq 4$, a regime where the matrix multiply will be bandwidth bound. We store the A matrix transposed to maximize bandwidth by using the widest possible vector loads while avoiding transposition after loading. Each warp computes four rows of output for all N output columns. Note that for $N \leq 4$ the B matrix fits entirely in the L1 cache. This scheme achieves 90 percent of peak bandwidth for $N \leq 4$ but starts to lose efficiency for larger N as the B matrix stops fitting into the L1 cache. Nonetheless, it continues to provide improved performance over existing libraries up to $N = 10$.

Figure 6.7 shows that our deployment kernel sustains a higher computational throughput than those from Nervana Systems [97] on the K1200 GPU, across the entire range of batch sizes that we use in deployment. Both our kernels and the Nervana kernels are significantly faster than NVIDIA CUBLAS version 7.0, more details are found here [34].

6.7.3 Beam Search

Performing the beam search involves repeated lookups in the n-gram language model, most of which translate to uncached reads from memory. The direct implementation of beam search means that each time-step dispatches one lookup per character for each beam. In Mandarin, this results in over 1M lookups per 40ms stride of speech data, which is too slow for deployment. To deal with this problem, we use a heuristic to further prune the beam search. Rather than considering all characters as viable additions to the beam, we only consider the fewest number of characters whose cumulative probability is at least p. In practice, we have found that $p = 0.99$ works well. Additionally, we limit ourselves to no more than 40 characters. This speeds up the Mandarin language model lookup time by a factor of 150x, and has a negligible effect on the CER (0.1-0.3% relative).

6.7.4 Results

We can deploy our system at low latency and high throughput without sacrificing much accuracy. On a held-out set of 2000 utterances, our research system achieves 5.81 character error rate whereas the deployed system achieves 6.10 character error rate. This is only a 5% relative degradation for the deployed system. In order to accomplish this, we employ a neural network architecture with low deployment latency, reduce the precision of our network to 16-bit, built a batching scheduler to more efficiently evaluate RNNs, and find a simple heuristic to reduce beam search cost. The model has five forward-only recurrent layers with 2560 hidden units, one row convolution layer (Section 6.3.7) with $\tau = 19$, and one fully-connected layer with 2560 hidden units. These techniques allow us to deploy Deep Speech at low cost to interactive applications.

6.8 Conclusion

End-to-end deep learning presents the exciting opportunity to improve speech recognition systems continually with increases in data and computation. Indeed, our results show that, compared to the previous incarnation, Deep Speech has significantly closed the gap in transcription performance with human workers by leveraging more data and larger models. Further, since the approach is highly generic, we've shown that it can quickly be applied to new languages. Creating high-performing recognizers for two very different languages, English and Mandarin, required essentially no expert knowledge of the languages. Finally, we have also shown that this approach can be efficiently deployed by batching user requests together on a GPU server, paving the way to deliver end-to-end Deep Learning technologies to users.

To achieve these results, we have explored various network architectures, finding several effective techniques: enhancements to numerical optimization through SortaGrad and Batch Normalization, evaluation of RNNs with larger strides with bigram outputs for English, searching through both bidirectional and unidirectional models. This exploration was powered by a well optimized, High Performance Computing inspired training system that allows us to train new, full-scale models on our large datasets in just a few days.

Overall, we believe our results confirm and exemplify the value of end-to-end Deep Learning methods for speech recognition in several settings. In those cases where our system is not already comparable to humans, the difference has fallen rapidly, largely because of application-agnostic Deep Learning techniques. We believe these techniques will continue to scale, and thus conclude that the vision of a single speech system that outperforms humans in most scenarios is imminently achievable.

Chapter 7

Keyword Spotting and Voice Activity Detection

7.1 Introduction

We propose a single neural network architecture to accomplish three tasks: large-vocabulary continuous speech recognition (LVCSR), on-line keyword spotting (KWS), and voice activity detection (VAD). The model is based on work in end-to-end speech recognition which uses the Connectionist Temporal Classification loss function coupled with deep Recurrent Neural Networks [41, 48]. In this work we develop the model and inference procedure for the KWS and VAD tasks. A thorough treatment of the benefits of this model for LVCSR is given in [2].

One of the main benefits in using the same architecture for all three tasks is simplicity. We need to maintain only a single architecture for training and deployment of the ASR, KWS and VAD model. We literally use the same model parameters for the KWS and VAD tasks. Given that these models run on-device, this can be a considerable space saving. However, as we show our model also achieves high accuracy on the VAD and KWS tasks.

Like in LVCSR, adopting a more end-to-end approach to the tasks of KWS and VAD also reduces the number of components needed to train the model. We are able to train LVCSR, KWS and VAD models without needing to bootstrap an alignment of output class labels to input frames. Furthermore we do not require the use of a pronunciation dictionary or models to build context-dependent output classes.

Many competing methods exist for KWS. Sometimes a lightweight model is needed which can run in real-time on a smart phone with a small memory footprint without draining the battery life. In other tasks a more heavyweight model can be used server side as an ‘audiogrep’ tool. In some cases the application needs to detect a single word or combination of words (e.g. “Ok Google”) with high accuracy. In other instances the application needs to search a list of words from a given vocabulary and allow the end-user to choose the keyword. In VAD, a super lightweight model is typically needed which can run as a background process on a sleeping smart phone for many hours without draining the battery. However, occasionally a larger model can be used on a plugged in device (e.g. the Amazon Echo) or server-side for end-pointing an utterance.

We specifically consider the task of on-line KWS on embedded devices - commonly referred to as a “wake word”. For the VAD we are mostly focused on determining when a user has finished speaking, a task often referred to as end-pointing. The constraints for this model are that it must fit on the smart phone and be able to run efficiently while the application is active. For the use-cases we consider this tends to be around an hour. However, the model we describe is general in the sense that the hyper-parameters can tuned such that one can trade-off accuracy and computational efficiency for any of the above tasks.

A model which fits our power and latency constraints (about 600K parameters) to be deployed on an iPhone 6 achieves a true positive rate of 94.9% for a 0.6% false positive rate in detecting the keyword “Olivia”. For the VAD task the same model achieves a 99.9% true positive rate at a 5.0% false positive rate when labelling a full utterance as containing speech or not.

7.2 Related Work

Recently, research in Automatic Speech Recognition (ASR) has trended towards more "end-to-end" models [48, 12, 18]. These models do not require an alignment of class labels to audio input frames to train. Often they output directly to characters [48] and even whole words [12] and read from the input waveform directly [142].

Here we explore the question of how to leverage this work for the tasks of Keyword Spotting and Voice Activity Detection. Intuitively, model architectures which work well for large vocabulary speech recognizers should also be useful for these simpler speech tasks.

Keyword spotting like LVCSR is generally done with an HMM to model the probability of the observation sequence given the hidden state sequence of the label [111, 112]. One can threshold the ratio of this probability to that of a filler HMM to tune false negatives for a given false positive rate. Like in LVCSR, one can substitute a DNN to compute the emission probabilities of the HMM instead of a GMM.

Recently, to solve the KWS task a variety of models have been proposed, most of which diverge from the usual ASR system. One method uses a bi-directional LSTM which outputs to whole keyword units [36]. The model is trained with CTC and does not need an alignment of the keyword to the audio input. However, this model has to be trained from scratch every time a new keyword is added or the wake word is changed. Furthermore, outputting to whole keyword units is not the most efficient use of the speech corpus. For example if we want to learn the keyword "okay sam" we would like to be able to leverage all utterances that have "okay" or "sam" present regardless if the two words occur consecutively. Another method uses a DNN to predict the words of a keyword. The data is aligned using a large vocabulary speech recognizer [13]. A custom scoring function operates on windows of outputs produced from the DNN in order to predict the presence of a key-word.

In these methods either an alignment is needed or the model outputs directly to keyword units. In contrast our model does not require the alignment of the transcript to the audio since we use the CTC cost function. Also since the model outputs to characters directly, we are able to better leverage the data and quickly adapt the model to new keywords.

Models for voice activity detection (VAD) have a wide range of complexity. Some simple and efficient techniques include a threshold on the energy of the audio signal, a threshold on the number of zero-crossings [66] or classifiers built from combinations of these features. Speech typically has higher energy than non-speech noise and fewer zero-crossings. These methods are typically not robust to non-stationary environments. More complex parameter estimation methods work better. A model resembling HMM-GMM ASR can be used. In this model a Gaussian represents the non-speech distribution and another represents the speech distribution [124]. An HMM is typically applied on top of these observation probabilities to smooth the estimates over time. A unique RNN architecture has also been shown to work well for VAD [60].

In the parameter estimation methods used for VAD, a corpus of audio frames labeled as speech or non-speech is needed. This corpus is typically produced with a forced-alignment from a large vocabulary ASR system.

7.3 Model

For a general keyword spotter we need a model which can give us $p(k \mid x)$ where x is a window of speech and k is any keyword. For VAD we use the same distribution and simply set k to the empty string.

We use the Connectionist Temporal Classification [40] (CTC) objective function to train an RNN on a corpus of utterance and transcription pairs. The CTC objective gives us the probability of any label string for a given utterance. We do not need an alignment here because CTC efficiently computes the score over all possible alignments. The objective function for an utterance X and corresponding transcription Y

is given by

$$p_{\text{ctc}}(Y \mid X) = \sum_{s \in \text{Align}(X,Y)} \prod_{t}^{T} p(s_t \mid X). \tag{7.1}$$

The Align(·) function computes the set of possible alignments of the transcription Y over the T time-steps of the utterance under the CTC operator. The CTC operator allows for repetitions of any character and insertions of the blank character, ϵ, which signifies no output at a given time-step.

For the on-line KWS task we must determine with low latency if the keyword has been said. Thus, in order to use this model for KWS we score a moving window of the audio so that we can find the keyword soon after it occurs. The score is computed as $p_{\text{ctc}}(k \mid x_{t:t+w})$ where k is any keyword and $x_{t:t+w}$ is a window of speech w frames long. For the VAD task we first compute the probability of no speech by setting k to the empty string. From this we can find the probability of speech by taking one minus the probability of no speech.

7.3.1 Network Architecture

The network accepts as input a spectrogram computed from the raw waveform sampled at 8kHz. The first layer is a 2-dimensional convolution with a stride of three [2]. For the next three layers of the network we use gated recurrent RNN layers [16] as they typically work as well as an LSTM but are more efficient [19]. The last layer is a single affine transformation followed by a softmax. The network outputs directly to characters in the alphabet including the blank and space characters.

7.3.2 Inference

In practice, as is typical of most KWS models, one has to choose a window size of speech over which to evaluate the possibility of the keyword being present. The accuracy of the algorithm can be sensitive to this parameter. If the window size is too small, then the entire keyword may not have been seen. On the other hand, if the window size is too large, other speech may be present and since our model evaluates the presence of only the keyword this can be problematic. We do not want to limit what the user can say directly before or after the keyword. In fact, allowing the user to talk-through the wake word without needing to pause between it and the actual utterance gives a more natural experience. Setting the window size parameter can be tricky given that the keyword can be realized in many different lengths depending on the rate of speech of the user.

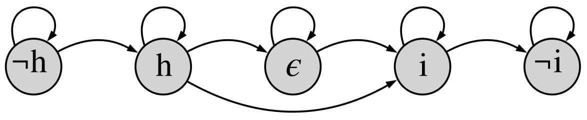


Figure 7.1: The keyword spotting state transition diagram for the keyword "hi".

To alleviate the sensitivity of the algorithm to the window size parameter we propose a modification to the CTC scoring algorithm presented above. For a given keyword k instead of scoring k under the model we instead score the regular expression $[\hat{}k_0]*k[\hat{}k_{n-1}]*$, where k_0 and k_{n-1} are the first and last characters of k. Figure 7.1 shows an example state transition diagram for this scoring function. An implementation for computing this modified CTC score is shown in Algorithm 2.

Algorithm 2 allows us to choose a window size long enough such that all occurrences of the keyword are shorter than it and not worry if other speech is present before or after the keyword.

Computing the VAD score is efficient as it reduces to summing the log probabilities of the blank character over the window of speech frames

$$\log p(\text{speech} \mid x_{t:t+w}) = 1 - \sum_{i=t}^{t+w} \log p_i(\epsilon \mid x_{t:t+w}). \tag{7.2}$$

Algorithm 2 Computing the score of a keyword, k, given the output probabilities of the RNN. The algorithm accepts l and P as parameters. The variable l is the keyword with ϵ inserted at the beginning, end and between every pair of characters of k. The matrix P contains the distributions over output characters in its columns for each time-step.

```
function ScoreKeyword($l$, $P$)
    $S \leftarrow$ size($l$)
    $T \leftarrow$ numberOfColumns($P$)
    $\alpha \leftarrow$ zeros(S, T)
    $\alpha_{1,1} \leftarrow 1 - P_{l_2,1}$
    $\alpha_{2,1} \leftarrow P_{l_2,1}$
    for $t = 2 : T$ do
        for $s = 1 : S$ do
            if $s = 1$ then
                $p \leftarrow 1 - P_{l_2,t}$
            else if $s = S$ then
                $p \leftarrow 1 - P_{l_{S-1},t}$
            else
                $p \leftarrow P_{l_s,t}$
            end if
            if $s > 2$ and $l_s \neq l_{s-2}$ and $l_s \neq \epsilon$ then
                $\alpha_{s,t} \leftarrow p * (\alpha_{s,t-1} + \alpha_{s-1,t-1} + \alpha_{s-2,t-1})$
            else if $s > 1$ then
                $\alpha_{s,t} \leftarrow p * (\alpha_{s,t-1} + \alpha_{s-1,t-1})$
            else
                $\alpha_{s,t} \leftarrow p * \alpha_{s,t-1}$
            end if
        end for
    end for
    return $\alpha_{S,T} + \alpha_{S-1,T}$
end function
```

7.4 Experiments

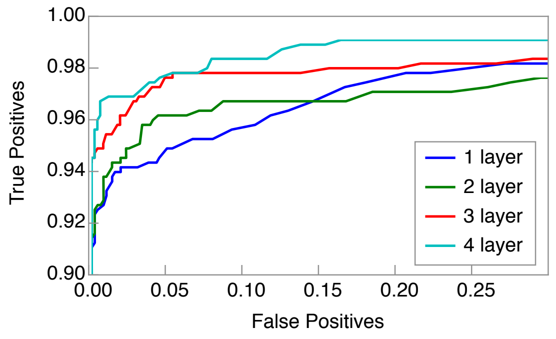
In this section we show several experiments evaluating the model for both the KWS and VAD task. We explore the performance of the model varying the number of layers, layer sizes and data used to train the system.

The model parameters are optimizes with stochastic gradient descent. All models are trained for 50 epochs where an epoch is a full pass through the data. The learning rate and momentum parameters are chosen to optimize speed of convergence. We anneal the learning rate by a factor of 0.9 every 5000 iterations. All experiments use a minibatch of 256 examples during training. We sort examples so that the minibatch consists of utterances of similar length for computational efficiency.

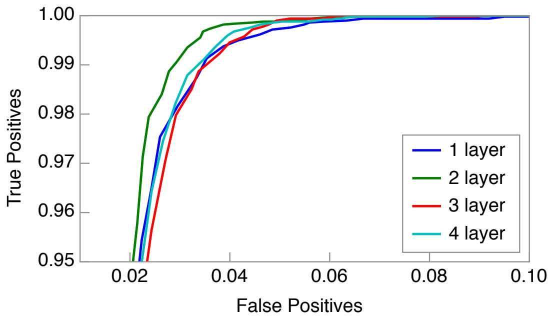
The architecture of the network is as described in Section 7.3.1. The filters for the convolution layer are 11 by 32 over the time and frequency dimensions respectively. We use 32 filters in all models.

The data used to train the model consists of two data-sets. The first data-set is a corpus 526K transcribed utterances collected on Android phones via an assistant like application. The second corpus consists of 1544 spoken examples of the name "Olivia", the keyword we want the model to recognize. The model is trained on both data-sets simultaneously. We do not need to pre-train on the large corpus prior to fine-tuning. We also use a collection of about a hundred hours of noise and music downloaded from the web to generate synthetic noisy examples of the keyword and empty noise clips. When training with the noisy data we replicate each keyword 10 times, each time with a random noise clip. We also use a corpus of 57K randomly sampled noise clips with a blank label as filler.

The KWS model is evaluated on a test set of 550 positive examples (e.g. containing the keyword "Olivia") and 5000 negative examples held-out from the large speech corpus described above. During inference we evaluate the utterance with Algorithm 2 every 100 milliseconds over a window of 800 milliseconds in order to detect the presence of the keyword. We classify an example as positive if the score found from the



(a) KWS



(b) VAD

Figure 7.2: The performance of the model on both the KWS and VAD tasks varying the number of hidden layers from 1 to 4. The layer size for all models is fixed at 256 hidden units.

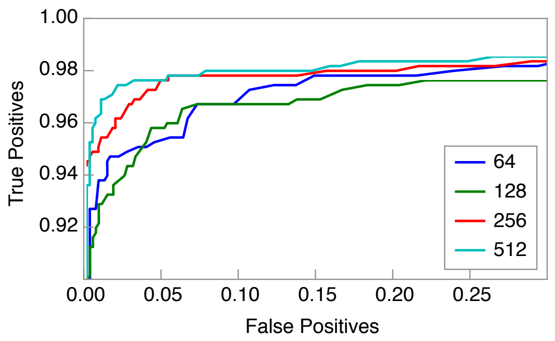
output of Algorithm 2 over the utterance is ever above a preset threshold. In the results shown below we plot ROC curves with the False Positive rate on the horizontal axis and the True Positive rate on the vertical axis.

We evaluate the same models on the VAD task. The positive examples are the same 5000 examples of speech used as the negative examples for the KWS task. We collected about 10 hours of non-speech audio from a variety of noise backgrounds. We sample 5000 random clips from the 10 hours of noise to construct the negative samples.

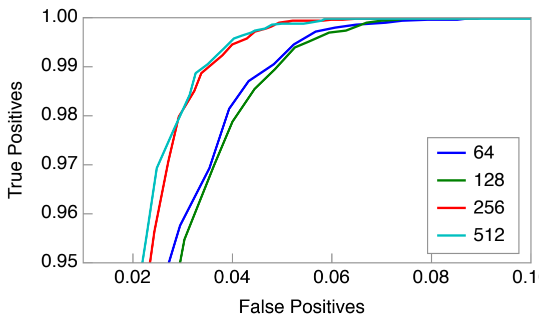
Figures 7.2 and 7.3 show that the model consistently improves at detecting the keyword as we increase the number of layers and the size of the model. In the KWS task we are constrained by computational resources. However, these results suggests that a promising direction for improving KWS performance is to find ways to deploy larger models more efficiently.

In the VAD task increasing the model depth does consistently improve performance. Similarly, increasing the layer size only helps up to a point. After the layers are larger than 128 units, the performance saturates. For most of the VAD models achieve near 99.9% true positive rate or higher at a fixed false positive rate of 5.0%.

In Figure 7.4 we see that adding noise to the keywords during training results in



(a) KWS



(b) VAD

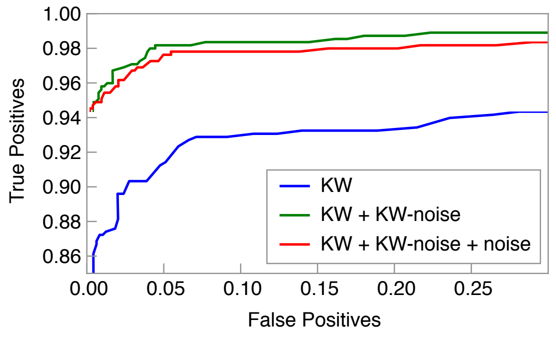
Figure 7.3: The performance of the model on both the KWS and VAD tasks varying the layer size from 64 to 512. The number of hidden layers for all models is fixed at 3.

substantial improvements. At a false positive rate of 1% the model with noise has a true positive rate of 95.8% compared to 87.2% for the model without noise. Further using the random noise data on its own does not help much; in fact the results are slightly worse. On the VAD task we also notice an improvement in the ROC curve as we add noise. However, the improvement in VAD is not nearly as significant as in KWS.

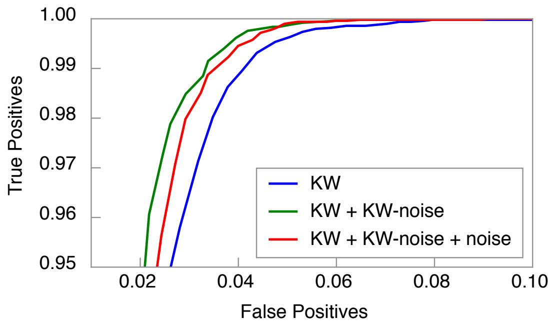
7.5 Conclusion

We have described a single neural network architecture which can be used for accurate large vocabulary speech recognition, key word spotting and voice activity detection. The model is simple to train and does not require an alignment or frame-wise labels. One only needs a large speech corpus with the corresponding transcriptions. We also give the details of an inference algorithm for KWS modified from the basic CTC scoring algorithm.

As our experiments have demonstrated, further improvement on the KWS task can come from better fitting the training data. Given the power and memory constraints of a deployed KWS and VAD model, finding ways to improve the efficiency of more expressive models will be a useful direction.



(a) KWS



(b) VAD

Figure 7.4: The performance of the model on both the KWS and VAD tasks as we add training data. The 'KW' model does not have any noise synthesis but does contain the 550K filler examples. The 'KW + KW noise' model includes the keyword data replicated 10 times with random noise. The 'KW + KW noise + noise' includes 57K random noise clips as filler.

Chapter 8

Conclusions

In this work we developed a framework for transcribing real-valued sequences with high accuracy. This framework has two primary axes for improving the generalization of the model. The first is the architecture of the model and the second is the amount of annotated data. As we increase the complexity of the model and the size of the dataset, efficient computation also becomes an essential ingredient.

We have shown how to apply this framework for three applications:

Arrhythmia Detection: We develop a model and dataset which can transcribe an ECG sequence into a sequence of arrhythmia predictions. The model's performance is comparable to that of board-certified cardiologists. Guided by our framework, we arrive at an architecture which consists of 34 layers with residual connections and batch normalization to make the optimization tractable. The dataset collected and annotated for the task contains more than 500 times the number of unique patients than previously studied corpora.

Speech Transcription: We develop a first-pass speech transcription model. By continuing to refine the model's architecture and grow the training datasets we arrive at the Deep Speech 2 recognizer. This model has error rates comparable to that of human transcribers in some English and Mandarin speech domains.

We create novel modules and learning strategies which improve the learning dynamics and generalization error of the model. We also develop a sophisticated data construction and augmentation pipeline in order to grow our dataset to unprecedentedly large sizes. Finally, in order to make the optimization and deployment of the model tractable, we develop several highly effective techniques for efficient computation.

Keyword Spotting and Voice Activity Detection: By approaching the problems of keyword spotting and voice activity detection in an end-to-end fashion, we are able to leverage most of the data and modeling techniques that we developed for the Deep Speech 2 model. With these techniques we develop a single model and a custom inference algorithm which is able to achieve high true positive rates at low false positive rates for both tasks. We deploy this model to a smart phone in real-time without any loss of accuracy.

Countless sequence transcription problems can benefit from the application of these techniques. However, there remains substantial room to improve along many fronts. As hardware and software for computing these models continues to improve rapidly, the key challenge will be our ability to learn models which generalize well. Collecting large annotated datasets is time-consuming, costly and in some cases impractical. Sample efficient methods which can still generalize well will make the application of these models much more accessible.

Nevertheless, the take home message of this work is that with sufficient resources we can achieve high accuracy on a wide range of sequence transcription problems. While not all problems can be solved this way, problems which do not require substantial high-level reasoning are typically amenable. For these problems we can leverage large annotated datasets, efficient computation and carefully tuned end-to-end models to match the performance of human experts on challenging tasks.

Bibliography

[1] Ossama Abdel-Hamid, Abdel rahman Mohamed, Hui Jang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *ICASSP*, 2012.

[2] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, JingDong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 173–182, 2016.

[3] Shane G Artis, RG Mark, and GB Moody. Detection of atrial fibrillation using artificial neural networks. In *Computers in Cardiology 1991, Proceedings.*, pages 173–176. IEEE, 1991.

[4] Yannis M. Assael, Brendan Shillingford, Shimon Whiteson, and Nando de Freitas. Lipnet: End-to-end sentence-level lipreading. nov 2016.

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

[6] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. End-to-end attention-based large vocabulary speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4945–4949. IEEE, 2016.

[7] Jon Barker, Ricard Marxer, Emmanuel Vincent, and Shinji Watanabe. The third chime speech separation and recognition challenge: Dataset, task and baselines. In *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*, pages 504–511. IEEE, 2015.

[8] Eric Battenberg, Jitong Chen, Rewon Child, Adam Coates, Yashesh Gaur, Yi Li, Hairong Liu, Sanjeev Satheesh, David Seetapun, Anuroop Sriram, and Zhenyao Zhu. Exploring neural transducers for end-to-end speech recognition. 2017.

[9] Sean Baxter. Modern gpu. *NVIDIA Research*, 2013.

[10] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[11] H. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, Norwell, MA, 1993.

[12] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *ICASSP*, 2016.

[13] Guoguo Chen, Carolina Parada, and Georg Heigold. Small-footprint keyword spotting using deep neural networks. In *ICASSP*, 2014.

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[15] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system.

[16] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.

[17] Jan Chorowski, Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. End-to-end continuous speech recognition using attention-based recurrent nn: first results. *arXiv preprint arXiv:1412.1602*, 2014.

[18] Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, KyungHyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In *NIPS*, 2015.

[19] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS Deep Learning Workshop*, 2014.

[20] Christopher Cieri, David Miller, and Kevin Walker. The fisher corpus: a resource for the next generations of speech-to-text. In *LREC*, volume 4, pages 69–71, 2004.

[21] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[22] Dan C Ciresan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *International Joint Conference on Artificial Intelligence*, pages 1237–1242, 2011.

[23] GD Clifford, CY Liu, B Moody, L Lehman, I Silva, Q Li, AEW Johnson, and RG Mark. Af classification from a short single lead ecg recording: The physionet computing in cardiology challenge 2017. 2017.

[24] Douglas A Coast, Richard M Stern, Gerald G Cano, and Stanley A Briller. An approach to cardiac arrhythmia analysis using hidden markov models. *IEEE*

Transactions on biomedical Engineering, 37(9):826–836, 1990.

[25] Adam Coates, Blake Carpenter, Carl Case, Sanjeev Satheesh, Bipin Suresh, Tao Wang, David J Wu, and Andrew Y Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 440–445. IEEE, 2011.

[26] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[27] Adam Coates, Andrew Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 215–223, 2011.

[28] G.E. Dahl, T.N. Sainath, and G.E. Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *ICASSP*, 2013.

[29] G.E. Dahl, D. Yu, and L. Deng. Large vocabulary continuous speech recognition with context-dependent dbn-hmms. In *ICASSP*, 2011.

[30] G.E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 2011.

[31] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[32] Dale Dubin. *Rapid Interpretation of EKG's.* USA: Cover Publishing Company, 1996, 1996.

[33] D. Ellis and N. Morgan. Size matters: An empirical study of neural network training for large vocabulary continuous speech recognition. In *ICASSP*, pages 1013–1016. IEEE, 1999.

[34] Erich Elsen. Optimizing RNN performance. `http://svail.github.io/rnn_perf`. Accessed: 2015-11-24.

[35] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, 2017.

[36] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *LREC*, volume 4, pages 69–71, 2004.

[37] Mark JF Gales, Anton Ragni, H AlDamarki, and C Gautier. Support vector machines for noise robust asr. In *Automatic Speech Recognition & Understanding, 2009. ASRU 2009. IEEE Workshop on*, pages 205–210. IEEE, 2009.

[38] X. Glorot, A. Bordes, and Y Bengio. Deep sparse rectifier networks. In *AISTATS*, pages 315–323, 2011.

[39] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[40] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *ICML*, pages 369–376. ACM, 2006.

[41] A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *ICML*, 2014.

[42] Alex Graves. Sequence transduction with recurrent neural networks. 2012.

[43] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385. 2012.

[44] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *ASRU*, pages 273–278. IEEE, 2013.

[45] Maya E Guglin and Deepak Thatai. Common errors in computer electrocardiogram interpretation. *International journal of cardiology*, 106(2):232–237, 2006.

[46] Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA*, 316(22):2402–2410, 2016.

[47] Awni Hannun. Sequence modeling with ctc. *Distill*, 2017. https://distill.pub/2017/ctc.

[48] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. abs/1412.5567, 2014.

[49] Awni Y Hannun, Andrew L Maas, Daniel Jurafsky, and Andrew Y Ng. First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns. *arXiv preprint arXiv:1408.2873*, 2014.

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[53] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. Scalable modified kneser-ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, 2013.

[54] Bo Hedén, Mattias Ohlsson, Holger Holst, Mattias Mjöman, Ralf Rittner, Olle Pahlm, Carsten Peterson, and Lars Edenbrandt. Detection of frequently overlooked electrocardiographic lead reversals using artificial neural networks. *The American journal of cardiology*, 78(5):600–604, 1996.

[55] G.E. Hinton, L. Deng, D. Yu, G.E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(November):82–97, 2012.

[56] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[57] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[58] De-An Huang, Li Fei-Fei, and Juan Carlos Niebles. Connectionist temporal modeling for weakly supervised action labeling. *European Conference on Computer Vision*, pages 137–153, jul 2016.

[59] Po-Sen Huang, Kshitiz Kumar, Chaojun Liu, Yifan Gong, and Li Deng. Predicting speech recognition confidence using deep learning with word identity

and score features. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 7413–7417. IEEE, 2013.

[60] Thad Hughes and Keir Mierle. Recurrent neural networks for voice activity detection. In *ICASSP*, pages 7378–7382, 2013.

[61] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[62] N. Jaitly, P. Nguyen, A. Senior, and V. Vanhoucke. Application of pretrained deep neural networks to large vocabulary speech recognition. In *INTERSPEECH*, 2012.

[63] Navdeep Jaitly and Geoffrey E Hinton. Vocal tract length perturbation (vtlp) improves speech recognition. In *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing*, 2013.

[64] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350, 2015.

[65] Jean-Claude Junqua. The lombard reflex and its role on human listeners and automatic speech recognizers. *The Journal of the Acoustical Society of America*, 93(1):510–524, 1993.

[66] Jean-Claude Junqua, Ben Reaves, and Brian Mak. A study of endpoint detection algorithms in adverse conditions: incidence on a dtw and hmm recognizer. In *EUROSPEECH*, 1991.

[67] Olga Kapralova, John Alex, Eugene Weinstein, Pedro J Moreno, and Olivier Siohan. A big data approach to acoustic model training corpus selection. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[68] David R Kincaid, Thomas C Oppe, and David M Young. Itpackv 2d user's guide. 1989.

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

[70] Brian Kingsbury, Tara N Sainath, and Hagen Soltau. Scalable minimum bayes risk training of deep neural network acoustic models using distributed hessian-free optimization. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[71] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[72] Tom Ko, Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. Audio augmentation for speech recognition. In *Interspeech*, 2015.

[73] Lingpeng Kong, Chris Dyer, and Noah A. Smith. Segmental recurrent neural networks. *ICLR*, nov 2016.

[74] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[75] Pablo Laguna, Roger G Mark, A Goldberg, and George B Moody. A database for evaluation of algorithms for measurement of qt and other waveform intervals in the ecg. In *Computers in Cardiology 1997*, pages 673–676. IEEE, 1997.

[76] César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 2657–2661. IEEE, 2016.

[77] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International*

Conference on, pages 8595–8598. IEEE, 2013.

[78] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[79] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[80] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–104. IEEE.

[81] Honglak Lee, Peter Pham, Yan Largman, and Andrew Y Ng. Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pages 1096–1104, 2009.

[82] Chris Lengerich and Awni Hannun. An end-to-end architecture for keyword spotting and voice activity detection. *NIPS 2016 End-to-End Learning for Speech and Audio Processing Workshop*, nov 2016.

[83] Cuiwei Li, Chongxun Zheng, and Changfeng Tai. Detection of ECG characteristic points using wavelet transforms. *IEEE Transactions on biomedical Engineering*, 42(1):21–28, 1995.

[84] Hairong Liu, Zhenyao Zhu, Xiangang Li, and Sanjeev Satheesh. Gram-ctc: Automatic unit selection and target decomposition for sequence labelling. *Proceedings of the 34th International Conference on Machine Learning*, feb 2017.

[85] Marcus Liwicki, Alex Graves, Horst Bunke, and Jrgen Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proceedings - 9th Int. Conf. on Document Analysis and Recognition*, volume 1, pages 367–371, 2007.

[86] Carla Lopes and Fernando Perdigo. Phone recognition on the timit database. *Speech Technologies*, 1:285–302, 2011.

[87] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. *Proc. ICML*, 30, 2013.

[88] Andrew Maas, Ziang Xie, Dan Jurafsky, and Andrew Ng. Lexicon-free conversational speech recognition with neural networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 345–354, 2015.

[89] Andrew L Maas, Awni Y Hannun, Christopher T Lengerich, Peng Qi, Daniel Jurafsky, and Andrew Y Ng. Increasing deep neural network acoustic model size for large vocabulary continuous speech recognition. *arXiv preprint arXiv:1406.7806*, 2014.

[90] Juan Pablo Martínez, Rute Almeida, Salvador Olmos, Ana Paula Rocha, and Pablo Laguna. A wavelet-based ECG delineator: evaluation on standard databases. *IEEE Transactions on biomedical engineering*, 51(4):570–581, 2004.

[91] SL Melo, LP Caloba, and J Nadal. Arrhythmia analysis using artificial neural network and decimated electrocardiographic data. In *Computers in Cardiology 2000*, pages 73–76. IEEE, 2000.

[92] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*, pages 167–174. IEEE, 2015.

[93] A. Mohamed, G.E. Dahl, and G.E. Hinton. Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, (99), 2011.

[94] George B Moody and Roger G Mark. A new method for detecting atrial fibrillation using RR intervals. *Computers in Cardiology*, 10(1):227–230, 1983.

[95] George B Moody and Roger G Mark. The impact of the MIT-BIH arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine*, 20(3):45–50, 2001.

[96] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[97] Nervana Systems. Nervana GPU. `https://github.com/NervanaSystems/nervanagpu`. Accessed: 2015-11-06.

[98] Jianwei Niu, Lei Xie, Lei Jia, and Na Hu. Context-dependent deep neural networks for commercial mandarin speech recognition applications. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2013 Asia-Pacific*, pages 1–5. IEEE, 2013.

[99] Chris Olah and Shan Carter. Attention and augmented recurrent neural networks. *Distill*, 2016.

[100] Christopher Olah. Understanding LSTM networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[101] Jiapu Pan and Willis J Tompkins. A real-time QRS detection algorithm. *IEEE transactions on biomedical engineering*, (3):230–236, 1985.

[102] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5206–5210. IEEE, 2015.

[103] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[104] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[105] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, K. Veselý, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, and G. Stemmer. The kaldi speech recognition toolkit. In *ASRU*, 2011.

[106] L.R. R. Rabiner. Tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):p257–286, 1989.

[107] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM, 2009.

[108] Pranav Rajpurkar, Awni Y Hannun, Masoumeh Haghpanahi, Codie Bourn, and Andrew Y Ng. Cardiologist-level arrhythmia detection with convolutional neural networks. *arXiv preprint arXiv:1707.01836*, 2017.

[109] S. Renals, N. Morgan, H. Bourlard, M. Cohen, and H. Franco. Connectionist probability estimators in hmm speech recognition. *IEEE Transactions on Speech and Audio Processing*, 2(1):161–174, 1994.

[110] Tony Robinson, Mike Hochberg, and Steve Renals. The use of recurrent neural networks in continuous speech recognition. In *Automatic speech and speaker recognition*, pages 233–258. Springer, 1996.

[111] J.R. Rohlicek, W. Russell, S. Roukos, and H. Gish. Continuous hidden markov modeling for speaker-independent wordspotting. In *ICASSP*, pages 627–71, 1990.

[112] J.R. Rohlicek, W. Russell, S. Roukos, and H. Gish. A hidden markov model based keyword recognition system. In *ICASSP*, pages 627–71, 1990.

[113] Tara N Sainath, Brian Kingsbury, Abdel-rahman Mohamed, George E Dahl, George Saon, Hagen Soltau, Tomas Beran, Aleksandr Y Aravkin, and Bhuvana Ramabhadran. Improvements to deep convolutional neural networks for lvcsr. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 315–320. IEEE, 2013.

[114] Tara N Sainath, Abdel-rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for lvcsr. In *Acoustics, speech and signal processing (ICASSP), 2013 IEEE international conference on*, pages 8614–8618. IEEE, 2013.

[115] Tara N Sainath, Oriol Vinyals, Andrew Senior, and Haşim Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4580–4584. IEEE, 2015.

[116] H. Sak, A. Senior, and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Interspeech*, 2014.

[117] H. Sak, O. Vinyals, G. Heigold, A. Senior, E. McDermott, R. Monga, and M. Mao. Sequence discriminative distributed training of long short-term memory recurrent neural networks. In *Interspeech*, 2014.

[118] Haşim Sak, Andrew Senior, Kanishka Rao, and Françoise Beaufays. Fast and accurate recurrent neural network acoustic models for speech recognition. *arXiv preprint arXiv:1507.06947*, 2015.

[119] Benjamin Sapp, Ashutosh Saxena, and Andrew Y Ng. A fast data collection and augmentation procedure for object recognition. 2008.

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

[121] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pages 437–440, 2011.

[122] Atman P Shah and Stanley A Rubin. Errors in the computerized electrocardiogram interpretation of cardiac rhythm. *Journal of electrocardiology*, 40(5):385–390, 2007.

[123] Jiulong Shan, Genqing Wu, Zhihong Hu, Xiliu Tang, Martin Jansche, and Pedro J Moreno. Search by voice in mandarin chinese. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[124] Jongseo Sohn, Nam Soo Kim, and Wonyong Sung. A statistical model-based voice activity detection. *IEEE Signal Processing Letters*, 6:1–3, 1999.

[125] Hagen Soltau, George Saon, and Tara N Sainath. Joint training of convolutional and non-convolutional neural networks. In *IEEE International Conference on Acoustic, Speech and Signal Processing*, 2014.

[126] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[127] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the Importance of Momentum and Initialization in Deep Learning. In *ICML*, 2013.

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

[129] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[130] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[131] Mintu P Turakhia, Donald D Hoang, Peter Zimetbaum, Jared D Miller, Victor F Froelicher, Uday N Kumar, Xiangyan Xu, Felix Yang, and Paul A Heidenreich. Diagnostic utility of a novel leadless arrhythmia monitoring device. *The American journal of cardiology*, 112(4):520–524, 2013.

[132] K. Veselỳ, A. Ghoshal, L. Burget, and D. Povey. Sequence-discriminative training of deep neural networks. In *INTERSPEECH*, pages 2345–2349, 2013.

[133] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. Phoneme recognition using time-delay neural networks. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(3):328–339, 1989.

[134] Chong Wang, Yining Wang, Po-Sen Huang, Abdelrahman Mohamed, Dengyong Zhou, and Li Deng. Sequence modeling via segmentations. feb 2017.

[135] Frank Wessel, Ralf Schluter, Klaus Macherey, and Hermann Ney. Confidence measures for large vocabulary continuous speech recognition. *IEEE Transactions on speech and audio processing*, 9(3):288–298, 2001.

[136] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for online training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

[137] P.C. Woodland and D. Povey. Large scale discriminative training of hidden markov models for speech recognition. *Computer Speech & Language*, 16(1):25–47, jan 2002.

[138] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256*, 2016.

[139] Takuya Yoshioka, Nobutaka Ito, Marc Delcroix, Atsunori Ogawa, Keisuke Kinoshita, Masakiyo Fujimoto, Chengzhu Yu, Wojciech J Fabian, Miquel Espi, Takuya Higuchi, et al. The ntt chime-3 system: Advances in speech enhancement and recognition for mobile multi-microphone devices. In *Automatic Speech*

Recognition and Understanding (ASRU), 2015 IEEE Workshop on, pages 436–443. IEEE, 2015.

[140] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

[141] M.D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q.V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G.E. Hinton. On rectified linear units for speech processing. In *ICASSP*, 2013.

[142] Zhenyao Zhu, Jesse H. Engel, and Awni Y. Hannun. Learning multiscale features directly from waveforms. In *Interspeech*, 2016.