Speech Representation Models for Speech Synthesis and Multimodal Speech Recognition

by

Felix Sun

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
James Glass
Senior Research Scientist
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Speech Representation Models for Speech Synthesis and Multimodal Speech Recognition

by

Felix Sun

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

The field of speech recognition has seen steady advances over the last two decades, leading to the accurate, real-time recognition systems available on mobile phones today. In this thesis, I apply speech modeling techniques developed for recognition to two other speech problems: speech synthesis and multimodal speech recognition with images. In both problems, there is a need to learn a relationship between speech sounds and another source of information. For speech synthesis, I show that using a neural network acoustic model results in a synthesizer that is more tolerant of noisy training data than previous work. For multimodal recognition, I show how information from images can be effectively integrated into the recognition search framework, resulting in improved accuracy when image data is available.

Thesis Supervisor: James Glass
Title: Senior Research Scientist

Acknowledgments

I want to thank everyone in the Spoken Language Systems group for such a great MEng experience. Thank you for showing me how exciting research can be, and giving me so much helpful advice. Special mentions to Dave for answering so many questions about Kaldi and speech recognition in general; to Ekapol for restarting the sls-titan-2 server every time I crashed it; and to Mandy for being a sounding board for all of my Torch bugs. I will miss eating lunch, discussing papers, and hanging out with all of you.

I want to thank Jim for being a wonderful advisor in every respect. At the beginning of last year, I had no idea what my thesis was going to be about - he guided me to interesting problems, and encouraged me to experiment and discover things on my own. Outside of the lab, I want to thank all of my friends at MIT for supporting me through a year that has very much been in flux.

Finally, I would like to thank Toyota, for sponsoring part of my thesis research.

Contents

Chapter 1

Introduction

Recently, the field of speech recognition has seen major advances. Applications that are commonplace today, like spoken dialog assistants on mobile devices, were considered firmly in the realm of science fiction 10 years ago. These applications are powered by general purpose speech recognizers that have steadily become more accurate through more than two decades of research. A speech recognizer is a holistic model of speech, relating snippets of raw sound to sentences in a language and taking into account acoustics, pronunciation, and grammar.

These advances in speech modeling have been used primarily for conventional speech recognition tasks, in which the goal is to decode a segment of speech in a given language. In this thesis, I adapt modern speech modeling techniques to two different problems: speech synthesis and multimodal speech recognition. Because speech synthesis is in some ways the inverse of speech recognition, the same speech models that are useful for recognition can be adapted for synthesis, with only small modifications. In the multimodal recognition problem, the speech to be decoded describes a given image, and the goal is to improve the accuracy of the recognizer using context from the image. This context can be added into the speech model through an image captioning neural network.

The rest of this thesis is organized as follows: Chapter 2 discusses the components of a speech recognition system. Chapter 3 introduces neural network acoustic models for speech. Chapter 4 describes the synthesis model in detail, including prior work on speech synthesis, system design, and results. Chapter 5 describes the multimodal recognizer. Chapter 6 offers concluding remarks and summarizes some engineering lessons learned.

Chapter 2

Background - An overview of speech modeling

Modern speech recognition systems consist of several models that are combined to make a transformation from raw audio to text. First, a *feature extractor* transforms raw audio into a feature vector at each of many short time intervals. Next, an *acoustic model* converts the features into a probability of the phone being pronounced in each timestep. This phone likelihood is converted into a probability distribution over sentences by a combination of a *pronunciation model*, which relates phones to words; and a *language model*, which assigns a prior probability to word sequences, based on the grammar of the language. A summary of this process is shown in Figure 2-1. Together, these models provide a comprehensive model of spoken language, using a common representation in terms of probability. In this thesis, I will modify these tools to perform two different tasks: speech synthesis and image-aided speech recognition.

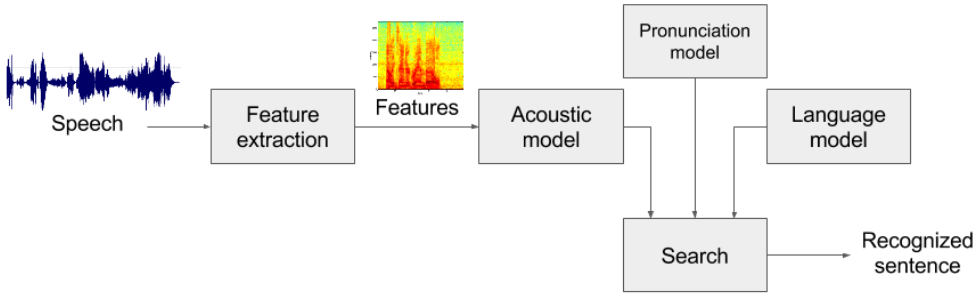

Figure 2-1: An overview of the major parts of a typical speech recognition system. Each of the gray boxes is discussed in a section of this chapter.

From the perspective of probability, a speech recognizer attempts to find the words W that maximize $P(W|S)$, the likelihood of the words given the observed speech. The variables W and S are related through the phone sequence P in a Markovian fashion: given the phones P, the words W and the speech S are independent. This formalizes the common-sense idea that the words only affect the speech through affecting the phones. With this assumption, the best words W^* for a speech segment can be expressed in the following way:

$$W^* = \operatorname*{argmax}_W P(W|S) \tag{2.1}$$

$$= \operatorname*{argmax}_W \frac{P(S|W) \cdot P(W)}{P(S)} \tag{2.2}$$

$$= \operatorname*{argmax}_W \sum_P \frac{P(S,P|W) \cdot P(W)}{P(S)} \tag{2.3}$$

$$= \operatorname*{argmax}_W \sum_P \frac{P(S|P,W) \cdot P(P|W) \cdot P(W)}{P(S)} \tag{2.4}$$

$$= \operatorname*{argmax}_W \sum_P \frac{P(S|P) \cdot P(P|W) \cdot P(W)}{P(S)} \tag{2.5}$$

$$= \operatorname*{argmax}_W \sum_P P(S|P) \cdot P(P|W) \cdot P(W) \tag{2.6}$$

$$\approx \operatorname*{argmax}_{W,P} P(S|P) \cdot P(P|W) \cdot P(W) \tag{2.7}$$

Here, $P(S|P)$ is the acoustic model, which describes the likely phone sequences for an utterance; $P(P|W)$ is the pronunciation model, which describes how a list of words ought to be pronounced; and $P(W)$ is the language model, which gives a prior over word sequences. In Equation 2.6, the normalization value $P(S)$ is dropped, because it is constant when selecting the best words for a particular S. Direct computation of Equation 2.6 is generally intractable, because of the large dimensionality of the search space W. Instead, state machines are used to represent the distribution, and approximate search algorithms are used to find the most likely W. Search often ignores the fact that different phone sequences can contribute to the same words, reducing the problem to finding the most likely joint phone and word sequence, as in Equation 2.7.

2.1 Speech feature extraction

Like all audio, recorded speech is a waveform, represented on a computer as a one-dimensional array containing the amplitude of the waveform at many closely-sampled points in time (e.g. 16 KHz or 8 KHz). Although some speech recognition systems learn to work with waveforms directly [1], the waveform is generally pre-processed into higher level features for most speech applications.

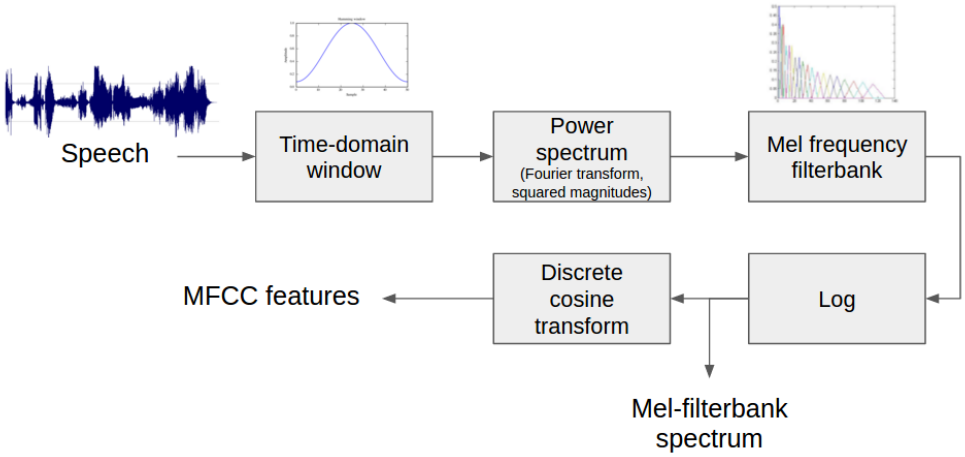

Figure 2-2: The MFCC extraction algorithm. The power spectrum of the speech is taken over a short window, and aggregated into 40 overlapping filters according to a frequency scale known as the Mel spectrum. A logarithm is taken, followed by a discrete cosine transform to make MFCC features for this timestep. This process is repeated, with the time-domain window shifted by 10 ms each time, to generate a series of features.

Mel-frequency cepstrum coefficients [2], or MFCCs, are the most popular feature representation for speech recognition. MFCCs summarize the amount of power a waveform has at different frequencies, over short (e.g. 25 ms) frames. To generate MFCCs for a single frame of speech, the Fourier transform of the waveform in the frame is first taken. The log power of the Fourier spectrum is then computed over 40 specific overlapping bins, each centered at a different frequency. This 40 dimensional vector is called the Mel-filterbank spectrum of the speech frame, and is sometimes used in place of MFCCs in speech recognition. To generate MFCCs, a discrete cosine transform is applied to these 40 power levels, and the magnitude of the first 13 responses is taken as the MFCC feature vector. This process is repeated for each frame of speech, to generate a matrix of features. The MFCC generation process is summarized in Figure 2-2. The feature vector at each timestep can also include deltas (derivatives) and double-deltas of each MFCC feature. Depending on the application, the MFCCs can be normalized at this stage, so they have zero mean and unit variance across each utterance, or per-speaker.

2.2 Acoustic models

An acoustic model computes the likelihood $P(s_{1:n}|p_{1:n})$ of some observed speech frames $s_{1:n}$, given per-frame phone labels $p_{1:n}$. It is typically trained on data that does not have per-frame phone labels. Instead, only the words are provided in most speech training data. A pronunciation dictionary (see Section 2.3) can convert the words into a sequence of phones $p_{1:m}$. (Throughout this document, I will use n to refer to the number of frames in a speech utterance, and m to refer to the number of phones or words in a sentence.) Traditionally,

GMM-HMM models (hidden Markov models with Gaussian mixture model emissions at each timestep) have been used as acoustic models. Recently, neural networks, discussed in Section 3, have eclipsed GMM-HMMs in acoustic modeling performance.

A GMM-HMM acoustic model [3] is a generative model of speech, which posits that each frame of speech is a sample from a probability distribution that is conditioned on the phone being pronounced (a Gaussian mixture model, or GMM). In turn, the phones are generated by another probability distribution, in which the identity of the phone at one frame depends on the identity of the phone at the previous frame (a hidden Markov model, or HMM). The joint probability of a sequence of phones $p_1...p_n$ and a sequence of speech frames $s_1...s_n$ is therefore

$$\begin{aligned} P(s_{1:n}, p_{1:n}) &= P(p_1) \cdot P(s_1|p_1) \cdot \prod_i P(p_i|p_{i-1}) \cdot P(s_i|p_i) \\ P(p_i|p_{i-1}) &\sim Multinomial(p_{i-1}; \theta_1) \\ P(s_i|p_i) &\sim GMM(s_i; \theta_2) \end{aligned} \tag{2.8}$$

In practice, the p's can have augmentations that make them more than just phone labels. For simplicity, we will refer to them as "phones", even though "states" may be a more accurate term.

The likelihood $P(s_{1:n}|p_{1:n})$ of the input speech frames assuming a particular phone sequence can be computed by multiplying the likelihoods of each $P(s_i|p_i)$, which is given directly by the relevant GMM. To estimate the parameters θ_1, θ_2 of a GMM-HMM model, the expectation-maximization (EM) algorithm is used on a collection of labeled training speech samples.

During training, the per-phone labels are not known, so the EM algorithm must maximize the likelihood of the phone list $P(s_{1:n}|q_{1:m})$ instead. This likelihood can be computed using dynamic programming over the frames of the speech sample:

$$\begin{aligned} P(s_{1:i}, p_i = q_j) =& P(s_{1:i-1}, p_{i-1} = q_j) \cdot P(s_i|q_j) \cdot P(q_j|q_j) \\ &+ P(s_{1:i-1}, p_{i-1} = q_{j-1}) \cdot P(s_i|q_j) \cdot P(q_j|q_{j-1}) \end{aligned} \tag{2.9}$$

With the correct initial conditions $P(s_1, p_1 = q_j)$, Equation 2.9 can be used to compute the likelihood of the first i speech frames, assuming that the i-th speech frame has phone label q_j. (This is a small modification of the forward-backward algorithm used to estimate hidden state likelihoods in an HMM.) The likelihood of the speech frames from $i+1$ to the end, $P(s_{i+1:n}, p_i = q_j)$, can be computed using an analogous recursion. The total probability that frame i has phone label q_j is simply the product of the two probabilities at frame

i:

$$P(s_{1:n}, p_i = q_j | p_{1:m}) = P(s_{1:i}, p_i = q_j | p_{1:m}) \cdot P(s_{i+1:n}, p_i = q_j | p_{1:m}) \tag{2.10}$$

The EM algorithm can use this information to estimate frame label frequencies.

The basic GMM-HMM system can be improved by using context-dependent p_i's, which allow the same phone to have different Gaussian mixtures, depending on the neighboring phones. In a context-dependent triphone model, each p_i is a concatenation of the current phone, the previous phone, and the next phone. This allows the model to better represent context-dependent variations in phone sound, like the fact that the /æ/ phone in "camp" is more nasalized than the /æ/ phone in "cap". A disadvantage of triphone labels is the abundance of labels required ($O(n^3)$ compared to $O(n)$), and therefore the abundance of parameters for the model. The number of labels can be reduced by clustering similar triphones into a single Gaussian mixture during the training process [4, 5].

Phones can also be split in the time dimension, to model the fact that a single phone can evolve over its duration. HMM phone models commonly assume that each phone is made of three sub-phones, each with a different Gaussian mixture, that must be pronounced in order [6]. Training a sub-phone model is simply a matter of replacing each phone in the corpus of labels with three sub-phones, then running GMM-HMM training normally.

2.3 Pronunciation models

The pronunciation model is responsible for computing a distribution over phones given words, $P(p_{1:n}|w_{1:m})$. Given a single sentence, this distribution is nearly deterministic, because each word can be pronounced in only a small number of ways, given by a pronunciation dictionary. Any time series of phones that collapses down to the correct sequence can be assigned a uniform probability. As such, the pronunciation model typically requires no machine learning methods.

For speech recognition, a pronunciation dictionary is usually a sufficient model for mapping words to phones. In other language modeling contexts, a richer model may be needed. For example, in mispronunciation detection and language tutoring systems, an inventory may be needed of the common *incorrect* ways to pronounce each word, which may be produced using clustering and nearest neighbor techniques [7]. In speech synthesis systems, the pronunciation model must maintain distributions over the length of each phone, as well as stress and prosody properties. (Speech synthesis pronunciation models are discussed in more detail in Section 4.1.)

2.4 Language models

The language model provides a prior distribution $P(w_{1:m})$ over which sequences of words are more likely to appear in a sentence. Typically trained on a large corpus of text, it allows the speech recognizer to distinguish between homophones and other words that are acoustically similar. The most common language model is the n-gram model [8], which assumes that the likelihood of a word at a particular location depends only on the words immediately preceding that location. For example, in a 3-gram model, the likelihood of a sentence $w_{1:m}$ is defined as

$$P_L(w_{1:m}) = \prod_i P(w_i|w_{i-1}, w_{i-2}) \tag{2.11}$$

In a n-gram model, the conditional probabilities of each word $P(w_i|w_{i-1}, \text{etc.})$ are parameters, which can be estimated by counting the occurrences of each n-gram combination in a large training corpus. A language model can then be evaluated by computing the model likelihood of a test set of sentences. Better language models will assign higher likelihood to the test sentences.

One challenge of training n-gram language models is that the number of parameters scales exponentially with n. The parameter $P(w_i|w_{i-k:i-1})$ needs to be estimated for every combination of $w_{i-k:i}$, of which there are $|\text{vocab}|^{k+1}$ such combinations. Furthermore, an n-gram probability cannot be estimated at all if that n-gram is not observed in the training corpus, and the estimate is likely to be poor if the n-gram is observed only a few times. To solve this problem, various "smoothing techniques" provide fallback options for n-grams that were not commonly seen in the training corpus. Such smoothing techniques, including Jelinek-Mercer smoothing and Kneser–Ney smoothing, give the n-gram probability as a linear combination of the empirical unigram fraction $\hat{P}(w_i)$, the empirical bigram probability $\hat{P}(w_i|w_{i-1})$, and so forth. The weight of each empirical n-gram depends on how much data was used to fit that n-gram.

2.5 Combining models using weighted finite state transducers

The acoustic, pronunciation, and language models must be combined to make a total speech recognition model. According to Equation 2.6, this should be done as follows:

$$P(w_{1:m}|s_{1:n}) \propto \sum_P P(s_{1:n}|p_{1:n}) \cdot P(p_{1:n}|w_{1:m}) \cdot P(w_{1:m}) \tag{2.12}$$

In general, exact computation of the most likely word sequence $\text{argmax}_w P(w_{1:m}|s_{1:n})$ is intractable, because of the exponential sizes of W and P. Instead, approximate search is performed using *weighted finite-state transducers* (WFSTs, see [9] for an overview). A WFST is a state machine, which defines a list of states,

including a start and end state. WFSTs define the score of an output sequence y given an input sequence x, making them natural for modeling conditional probabilities. The score is defined over state transitions induced by the input sequence: The transducer starts in the start state, and each input character causes a state transition, with an associated score. Each state transition also produces an output character. This transition process is defined by the transition function $\mathbf{T}$: $score = \mathbf{T}(s_i, x_i, y_i, s_{i+1})$. If the end state is reached, the transducer resets into the start state. The transducer continues to accept inputs and produce outputs until the input sequence terminates. Throughout this thesis (and in speech recognition in general), the scores will represent log likelihoods, so an impossible state transition can be represented with a score of $-\infty$.

Formally, a WFST is fully specified by a list of states s, the transition function $\mathbf{T}$, and initial and final states. Assuming that each (s_i, x_i, y_i) tuple implies only one possible s_{i+1}, the score for an output sequence given an input sequence is simply the sum of the scores for each transition required to generate the output sequence.[1] In speech recognition WFSTs, the scores represent log probabilities, so this corresponds to multiplying together the probabilities of each step in the sequence.

2.5.1 WFSTs for acoustic models

WFSTs for acoustic models represent $P(S|P)$ for a single known speech audio sequence S - they assign likelihood scores to phone sequences based on how well they reflect the speech. Because S is fixed, the WFST accepts no inputs, and produces phone sequences as output. A WFST built from a GMM-HMM model has one state per timestep, with one transition from timestep i to timestep $i+1$ for each phone p in the alphabet that also emits p as output. The score of the transition is the log likelihood of speech frame i assuming the current phone is p, as computed from the GMM. Therefore, the score of an output sequence is the total log likelihood of that phone sequence. An example of such a WFST is shown in Figure 2-3.

If the acoustic model HMM uses context-dependent or sub-phone states, these states can be compacted into context-independent phones during WFST creation. Often, the mapping between states and phones is represented using a separate context WFST.

2.5.2 WFSTs for pronunciation models

A pronunciation model can also be expressed as a WFST, whose state transitions accept phones and produce words. The pronunciation WFST has one state for each possible phone prefix in the vocabulary. From state

[1] If this condition is not true, then multiple paths through the state space can result in the same output sequence. In this case, we need a way to combine scores in parallel as well as in series to compute a total output score. If the WFST represents $\log P(Y|X)$, this is done with log-addition. However, if the WFST represents $\log P(X|Y)$, there is no direct way to calculate the total likelihood of Y.

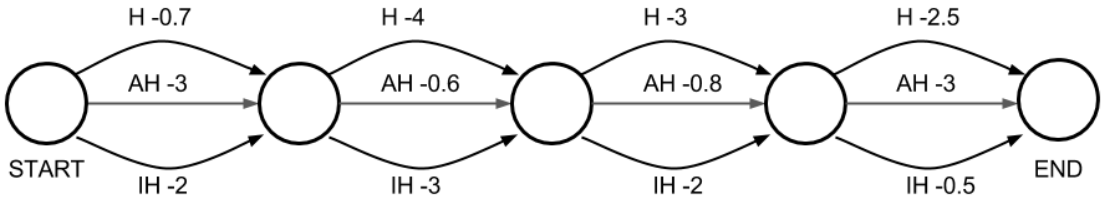

Figure 2-3: Illustrations of an acoustic WFST for a toy utterance, generated from a GMM-HMM model. Each state represents a timestep, and each transition emits a phone with a score corresponding to the likelihood of that phone at that timestep. The total log likelihood of a phone sequence is equal to the sum of the scores of each transition in the sequence: for example, "H AH AH IH" is the most likely sequence in this WFST, with a log likelihood of -2.6.

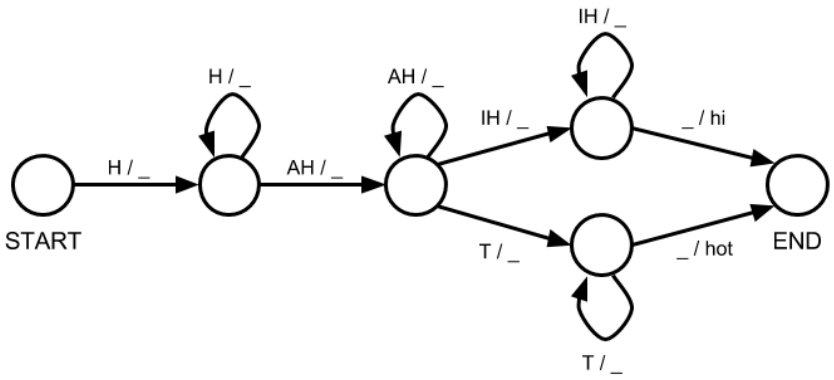

Figure 2-4: A toy pronunciation model, represented as a WFST. The transitions are expressed in the form [input] / [output], with _ representing an empty (epsilon) character. All transition scores are 1.0, because this particular pronunciation model is deterministic: given a set of phones, there is only one word that could be pronounced. The self-loops allow each phone to be an arbitrary length.

s, it accepts as input any phone that forms a valid prefix when appended to the prefix represented by s. Any state whose phones spell a valid word also accepts a blank input, which causes the WFST to emit the word represented by the state and transition to the end state. An example of such a pronunciation WFST for a very simple vocabulary is shown in Figure 2-4. Because most pronunciation models are deterministic, all state transitions have the same unit score. This implies that $P(P|W) = k$ for all sequences P that spell out W.

2.5.3 Composing WFSTs

The acoustic and pronunciation WFSTs can be combined using a procedure called *WFST composition*, to compute

$$P(s_{1:n}|w_{1:m}) = \sum_{\text{all } P} P(s_{1:n}|p_{1:n}) \cdot P(p_{1:n}|w_{1:m})$$

. (In the log domain, this is instead a log-sum over the sum of the two log-likelihoods.) A state-of-the-art algorithm for WFST composition is given in [10]. At a high level, the states of the composed WFST are (s_1, s_2), where s_1 is a state in the first WFST, and s_2 is a state in the second WFST. The transition function takes an input from the input domain of the first WFST, uses the first WFST's transition function to generate a new s_1' and an input to the second WFST, and then uses the second WFST's transition function to generate a new s_2' and an output. It returns (s_1', s_2'), the generated output, and the sum of the scores of the two inner transition functions.

Not every tuple (s_1, s_2) is necessary in the composed WFST, however. Some composed states are not reachable, in that no sequence of inputs will land the first WFST in state s_1 and simultaneously land the second WFST in state s_2. For example, if we compose the toy acoustic and pronunciation models in Figures 2-3 and 2-4, the composite state consisting of the first timestep from the acoustic WFST and the "AH" self-loop from the pronunciation WFST is not reachable. This suggests a WFST composition algorithm that resembles flood-filling: a horizon list of composite states is maintained, which is initialized with $(s_{1-start}, s_{2-start})$, the starting state of the composite transducer. The horizon is repeatedly expanded by considering all possible inputs at a horizon state, and adding any newly discovered composite states to the horizon. This also enumerates all of the transitions possible in the composite WFST.

2.5.4 Language model WFSTs

A language model is a distribution over sequences of words $P(w_{1:m})$. Therefore, a WFST representation of a language model simply assigns a score to each input sequence, without making any changes: for every transition, the input is the same word as the output, but the score may vary depending on input and state.

An n-gram language model WFST has one state for every combination of words $(w_1, w_2, ..., w_{n-1})$ represented by the model. The score for transitioning from state $(w_{1:n-1})$ to state $(w_{2:n})$ is equal to $\log P(w_n|w_{1:n-1})$, as defined by the n-gram model. Discounted n-gram models, which use a linear combination of smaller m-gram models for each w_n, can also be represented with a separate set of m-gram states for word combinations that don't have a full n-gram probability.

A bigram language model WFST is shown in Figure 2-5. Each state represents a word in the vocabulary; the wildcard * state represents any word that is not taken by some other state. The score of a transition from w_1 to w_2 is $\log P(w_2|w_1)$. Each transition accepts and emits w_2, the word of the target state. This way, the total score for a particular sentence is the log likelihood of that sentence. Note that some discounted unigram probabilities are embedded in this WFST, in the outbound transitions from the wildcard state. In particular, $\log P(w_i = \text{"like"}) = -4.5$, as long as the previous word isn't "I".

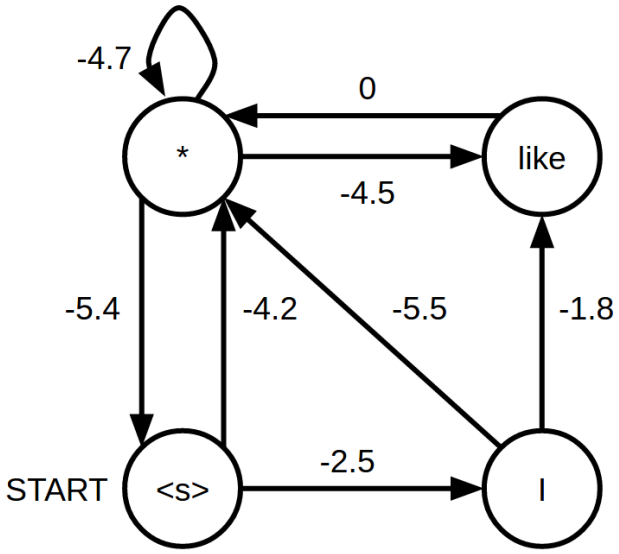

Figure 2-5: A bigram language model that prefers phrases beginning with "I like...". Only the score for each state transition is shown: the input for each transition is the target of the arrow, and the output is the same as the input.

To compute the total $P(W|S)$, the language model is composed with the acoustic-plus-pronunciation model. A Viterbi-like decoding algorithm can then be run on the composite WFST, yielding the highest-scoring sequence of words, conditioned on the speech. In practice, the language and acoustic models can be composed together during training time, because neither model is dependent on the utterance to be decoded. The overall WFST composition procedure is often described as

$$H \circ min(det(L \circ G)) \tag{2.13}$$

, where H is the acoustic WFST ("HMM"), L is the pronunciation WFST ("lexicon"), G is the language model WFST ("grammar"), and $\circ$ represents the composition operation. After L is composed with G, the resulting WFST is determinized and minimized, two operations that simplify and standardize the transducer without changing its meaning.

Chapter 3

Neural models for speech recognition

Neural networks are a class of machine learning models that can learn a complex relationship between a set of input values and output values. Recently, they have begun replacing GMM-HMMs in acoustic modeling, in which the relationship between MFCCs and phone labels is complex and non-linear. This chapter will introduce deep neural networks in Section 3.1, and recurrent neural networks in Section 3.2. Both of these networks model the relationship between MFCC frames and per-frame phone labels; they are analogous to the GMM part of the GMM-HMM. To relate per-frame phone labels to word-level phone labels (and thus avoid the need for training data with per-frame labels), a *sequence to sequence mapping* must be used. Section 3.3 discusses the connectionist temporal classifier and Section 3.4 discusses attention-based decoding, two such mapping strategies.

3.1 Neural networks and DNNs

Neural networks are a class of function approximators $y = f(x; \theta)$, whose parameters θ can be adjusted using exemplar samples of x and y (which can both be vectors of arbitrary size). Neural networks are composed of units that we will call *neurons*, which represent a simple function from x to a scalar y_i. This function is simply a linear transformation with a soft clipping function:

$$y_i = \sigma(w \cdot x + b) \tag{3.1}$$

The operator σ represents an element-wise sigmoid function: $\sigma(x) = 1/(1 + e^{-x})$, which limits the output to the range $(-1, 1)$ in a smooth way. Other non-linearities may also be used. W (a vector of size equal to the size of x) and b (a scalar) are the two parameters of this neuron.

Given a training point (x_{tr}, y_{tr}), the parameters of the neuron can be updated so that the neuron produces an output that is close to y_{tr} when it receives x_{tr} as input. This is done with a gradient descent algorithm:

$$Err = (y_{tr} - y_i)^2 \tag{3.2}$$

$$\frac{\partial Err}{\partial w} = \frac{\partial}{\partial w}(y_{tr} - \sigma(w \cdot x_{tr} + b))^2 \tag{3.3}$$

$$w \leftarrow w - \alpha \cdot \frac{\partial Err}{\partial w} \tag{3.4}$$

α is a learning rate parameter. This update step will change w in the direction of the negative error gradient, to move w towards a value that minimizes the error. A similar operation can be done for b. By repeatedly updating the parameters over a training set, the neuron can be made to approximate the training set.

Equation 3.2 defines the *loss* of the neural net, which is the objective, as a function of the expected output and the actual output, that training attempts to minimize. Squared error, which is used in this example, is often suitable for learning continuous y. If y is a binary variable, cross-entropy loss may be more suitable:

$$L_{cross-entropy}(y_{tr}, y_{out}) = y_{tr} \log y_{out} + (1 - y_{tr}) \log (1 - y_{out}) \tag{3.5}$$

One neuron can only model a very limited set of functions: the set of clipped hyperplanes. More complex function approximators can be made by stacking layers of multiple neurons. The output of one neuron serves as part of the input for the next layer of neurons:

$$\begin{aligned} x_1 &= \sigma(W_1 \cdot x + b_1) \\ x_2 &= \sigma(W_2 \cdot x_1 + b_2) \\ y &= \sigma(W_3 \cdot x_2 + b_3) \end{aligned} \tag{3.6}$$

Here, each W is a matrix, and each b is a vector. Each W and b represents one layer, which has multiple neurons in parallel: the matrix multiplication is equivalent to computing a vector of neurons, each with the same input and a different output. This configuration of stacked neurons, densely connected by layer, is called a *deep neural network* (DNN). Such networks can have varying depths and layer sizes (the size of each intermediate x_i).

DNNs can represent highly non-linear relationships between x and y, if trained with enough training data. Training a DNN is analogous to training a single neuron: the derivative of the error with respect to each parameter is used to perturb the parameter. These derivatives can be computed algorithmically for very large networks, and the updates can be performed in parallel on a GPU.

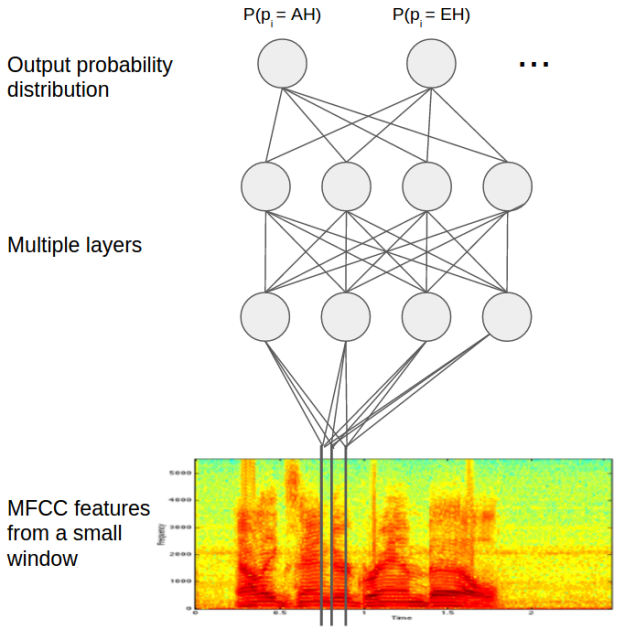

Figure 3-1: A frame-labeling DNN, which accepts MFCC features from a small window of frames, and outputs a probability distribution over phone labels for a particular frame. (Note that spectrogram images are used to represent MFCC data in diagrams throughout this thesis.)

After a slowdown in research on neural networks for speech, DNNs were the first new neural architecture to be successfully applied to recognition. DNNs can directly replace GMMs in an acoustic modeling framework [11]: a DNN can be trained to map MFCC features from a frame, plus MFCC features from surrounding frames, into a probability distribution over the phone label at the frame. The resulting probability distribution can be used as the acoustic likelihood in an HMM. An example of such an architecture is shown in Figure 3-1.

DNN models can only be trained with per-frame phone labels, which can be obtained by training a GMM-HMM model first, and force aligning the training data with the trained model. Sometimes, DNN models are *pretrained* to bias the parameters of the model in a meaningful way, before gradient descent is used. In restricted Boltzmann machine (RBM) pretraining [12], which is used in [11], each layer of the DNN is used in isolation to build an autoencoder model, whose goal is to reconstruct the input. The layers are trained to perform the autoencoding task, and then trained to perform the real classification task.

3.2 RNN models (Long short-term memory networks)

DNN models often lack an understanding of the time dimension. While they can classify phones in a context-dependent way, they do not explicitly model temporal dynamics; rather, features from all of the timesteps

are combined together into one input vector. Recurrent neural networks (RNNs), in contrast, process inputs one step at a time, and are explicitly symmetric in the time dimension. Therefore, some recent systems [13, 14] have had success in using RNNs to train acoustic models.

Recurrent neural networks (RNNs) model a mapping from $x_{1:n}$ (a vector x of n elements) to $y_{1:n}$. In general, the mapping is of the form

$$\begin{aligned} y_t &= f_1(x_i, s_{t-1}; \theta) \\ s_t &= f_2(x_i, s_{t-1}; \theta) \end{aligned} \tag{3.7}$$

for some functions f_1 and f_2 parameterized by θ. RNNs can be thought of as state machines that processes an input to create an output and transduce a state update in each timestep. In the language of DNNs, each timestep update can be thought of as a DNN: to process a sequence of inputs, an RNN uses the same neural network n times to transform each element of the input into an element of the output. It also uses a different neural network to update a memory variable in between timesteps. This combination of symmetry across timesteps and a state variable allows RNNs to model complex time dynamics using relatively fewer parameters than a DNN.

Long short-term memory (LSTM) networks [15] are a type of RNN often used in speech recognition applications. In an LSTM, the state transition and output functions are broken into a number of steps:

$$i_t = \sigma(W_{ix}x_t + W_{iy}y_{t-1} + W_{ic}c_{t-1} + b_i) \tag{3.8}$$

$$f_t = \sigma(W_{fx}x_t + W_{fy}y_{t-1} + W_{fc}c_{t-1} + b_f) \tag{3.9}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_{cx}x_t + W_{cy}y_{t-1} + b_c) \tag{3.10}$$

$$o_t = \sigma(W_{ox}x_t + W_{oy}y_{t-1} + W_{oc}c_t + b_o) \tag{3.11}$$

$$y_t = o_t \circ \tanh(c_t) \tag{3.12}$$

The process is also summarized in Figure 3-2.

In this model, the W and b variables are the parameters - each W represents a matrix, and each b represents a vector. The variables c_t and y_t together represent the state of the recurrent model. The state variables, as well as the intermediate results f_t and o_t, are all vectors of the same size - usually the output vector length $|y_t|$. The symbol $\circ$ represents element-wise multiplication.

The state variable c_t can be thought of as the "long-term memory" portion of the network - it is copied over from the previous value c_{t-1}, pending control by the "forget network" value f_t, which zeros out some

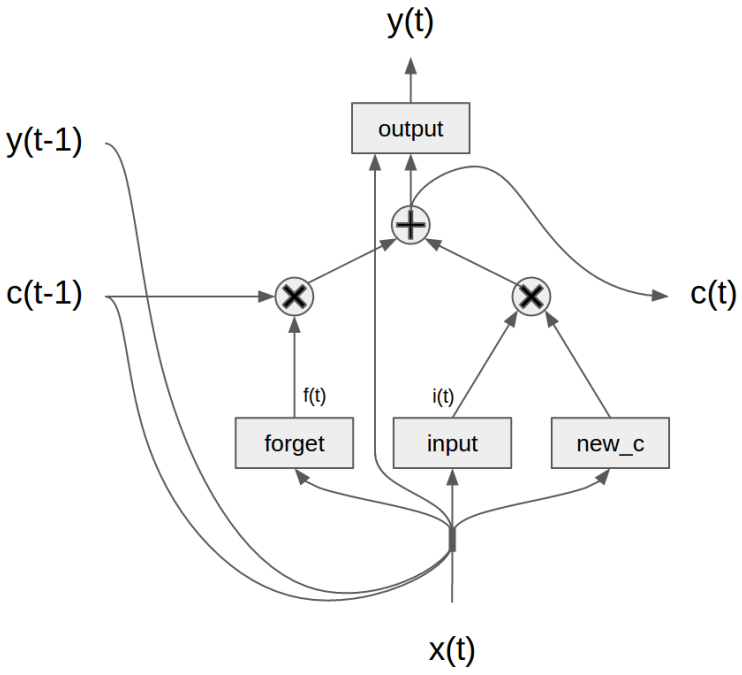

Figure 3-2: Data flow within one timestep of an LSTM. The gray boxes represent concatenating all of the inputs together, followed by multiplication by a parameter matrix, offset by a parameter vector, and a non-linear operation. Technically, the "output" operation does not depend c_{t-1}, but this is ignored to simplify the diagram.

of the elements of c_{t-1}. The long-term memory is designed to capture the effects that an x from many timesteps ago may have on y. The LSTM is usually initialized so that $f(t)$ is the one vector, encouraging the network to keep c values for many timesteps.

Parts of c_t can be rewritten at each timestep with new values, shown by the tanh function in Equation 3.10 and by the "new_c" network in Figure 3-2. Some of the new values are also blocked by an input network vector i_t, which functions analogously to the forget network vector f_t. Finally, the current long-term state c_t and the inputs to the current timestep are used by the output network to make the output, y_t. y_t is also used as a state variable, albeit one that is created from scratch at every timestep.

Using stochastic gradient descent, parameters can be found that map X to Y in a training set. In a speech recognition LSTM network (like in [16]), X is the sequence of MFCC features for a sentence, and Y represents a probability distribution over the phone identity of the speech frame at the current timestep. In these cases, each y_i is a vector of length equal to the size of the phone alphabet, and the activation at each element of the vector is interpreted as a probability. Such a network can then be trained with speech that has been labeled frame-by-frame. Two LSTM layers can also be stacked on top of each other, with the output of the first layer feeding into the input of the second layer. One such architecture, described in [16],

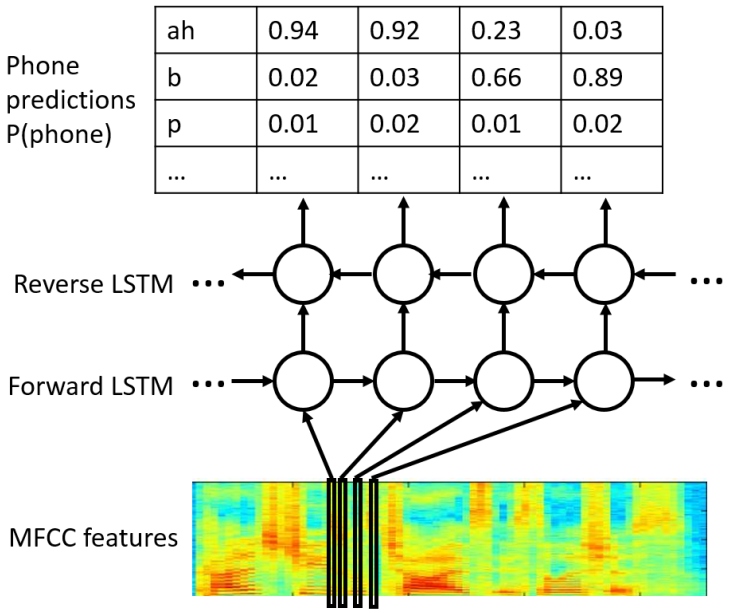

Figure 3-3: A bi-directional LSTM phone recognizer, as described in [16]. MFCC features are used as the input to two LSTM layers, one propagating information forwards, and one backwards. The result is interpreted as a probability distribution over the phone being pronounced in each frame.

is shown in Figure 3-3.

This LSTM model is not directly compatible with the probabilistic framework described in the beginning of this section, because the LSTM model produces $P(P|S)$, while the Equation 2.6 expects $P(S|P)$. Bayes' rule can be used to relate the two quantities:

$$P(S|P) = \frac{P(P|S) \cdot P(S)}{P(P)} \propto \frac{P(P|S)}{P(P)} \tag{3.13}$$

($P(S)$ does not matter in a speech recognition context, because we are not comparing between different values of S.) The prior distribution over phones $P(P)$ is estimated by soft-counting the fraction of frames occupied by each phone in the training data [11]. The training data is fed through the trained LSTM model to compute the posterior distribution $P(P|S)$ for each frame. The probability of a phone is then computed as the sum of the posterior probabilities of the phone in each frame, divided by the number of frames.

The stacked LSTM model can be trained to recognize speech at the frame level. However, speech recognition training data does not typically contain frame-level labels, though some datasets, like TIMIT [17], are an exception. Much more commonly, speech is labeled at the word level - each audio file of a sentence is labeled with the words spoken in the sentence, without any timing information. For LSTM models to be compatible with word transcripts, there must be a mapping between two sequences of different lengths: the sequence of per-frame phone labels that the LSTM outputs, and the sequence of words (or

phones) that is provided in the training data. (In a GMM-HMM acoustic model, this is the responsibility of the HMM.) Two such mappings are used in speech recognition: connectionist temporal classification (CTC), and attention-based models.

3.3 Connectionist temporal classification

Connectionist temporal classification (CTC) [18] is an objective function that is applied to the output of a neural net, similar in function (if not in form) to cross-entropy loss or squared loss. It measures the difference between a sequence of labels with one element per frame, and a ground truth sequence of labels. It assumes that the the per-frame output of the neural net is allowed to have repeated labels and blanks, when compared to the ground truth sequence. For example, "hhh---e--l-ll-o-o" is allowed to correspond to the ground truth sequence "hello".

Formally, the CTC objective function is defined as follows. Let P be the ground truth sequence of labels, with a blank character inserted between every element of the sequence, before the first character, and after the last character. If the original ground truth sequence is of length l, then P is of length $2l + 1$. A frame sequence $y_{1:n}$ is valid for P (denoted as $y_{1:n} \in V(P)$) if every element p of P can be mapped to a non-empty, contiguous, and in-order subsequence of $y_{1:n}$ that consists only of at least one s character and some (possibly zero) blanks. All of the the subsequences together must encompass all of $y_{1:n}$, without any gaps. This allows the "hello" example above to be correctly mapped, but does not permit y sequences with missing characters ("h-e-----o") or extra characters ("h-ee--ll-g-l-o").

The CTC score of a particular speech input is the sum of the probabilities that the speech input maps to frame-level label $y_{1:n}$, for all $y_{1:n} \in V(P)$. If the LSTM layer described in the previous section gives $P_t(y)$, the probability that the t-th frame has label y for all t and y, then the total CTC probability is

$$P_{CTC} = \sum_{y_{1:n} \in V(P)} \prod_t P_t(y_t) \tag{3.14}$$

This probability can be efficiently calculated in a dynamic programming fashion. A series of subproblems $P_{CTC}(i, t)$, corresponding to the probability that the first t elements of the input sequence correspond to the first i elements of P, can be computed recursively.

The CTC score of a neural net on some input can be maximized using stochastic gradient descent, because the CTC score is a purely algebraic mapping of some matrix of frame probabilities to a single number. Therefore, RNN models, like those described in the previous section, can be trained with a CTC objective function, as was done in [14].

3.4 Attention-based decoders

Attention-based models are a more general solution to the problem of using neural nets to generate a sequence of elements. In these models, the output sequence is produced by an RNN which takes a "glimpse" of the input sequence - a controlled subsample of the input. In addition to producing an output, the RNN is also responsible for choosing the next glimpse it takes of the input sequence. This process can continue for any number of steps, allowing for a fully flexible mapping from one sequence to another through a series of glimpses. This approach is applied to image recognition - mapping a "sequence" of pixels to a sequence of items in the image - in [19]; to image description using natural language in [20]; and to speech recognition in the Listen, Attend and Spell system [21]. The following description of an attention-based sequence mapping is based on the one described in [21].

In an attention-based model, there is some input sequence $h_{1:n}$ that must be mapped to some output sequence $y_{1:m}$, for different lengths m and n. In speech recognition networks, h is usually the output of an RNN, as described in the first section. At each output timestep t, a glimpse of the input sequence is taken. This glimpse has the same dimensions a single element of the input, and is made of a linear combination of the inputs, whose weights depend on some trainable function f_{weight}:

$$g_t = \sum_i h_i \cdot f_{weight}(h_i, w\_state_t) \tag{3.15}$$

The weighting function f_{weight} can be any neural network; for example, in the Listen, Attend and Spell system, it is a single dense layer with a softmax activation to ensure that the weights of all n elements of the input sum to 1.

The glimpses g_t are consumed and the w_state's are produced by a decoder RNN, which also produces the output sequence $y_{1:m}$. The entire mapping from h to y is summarized in Figure 3-4, which shows the process of generating a single y_t.

Like the CTC model, attention layers are fully differentiable, so they can be trained using gradient descent. In fact, an entire speech recognition model - that extracts features from MFCC frames using an LSTM, performs an attention remapping on these features, and then outputs phone labels using another LSTM - can be trained together using gradient descent. During training, the decoder is run m times, to generate a $y_{1:m}$ that can be compared against the ground truth. During testing (and deployment), the output sequence length m is not known, so a method is needed to determine how many iterations the decoder should be run. (In theory, the decoder can be run an arbitrary number of times for the same input, because the decoder chooses a new glimpse of the input at each step, and therefore never "runs out" of input.) Usually,

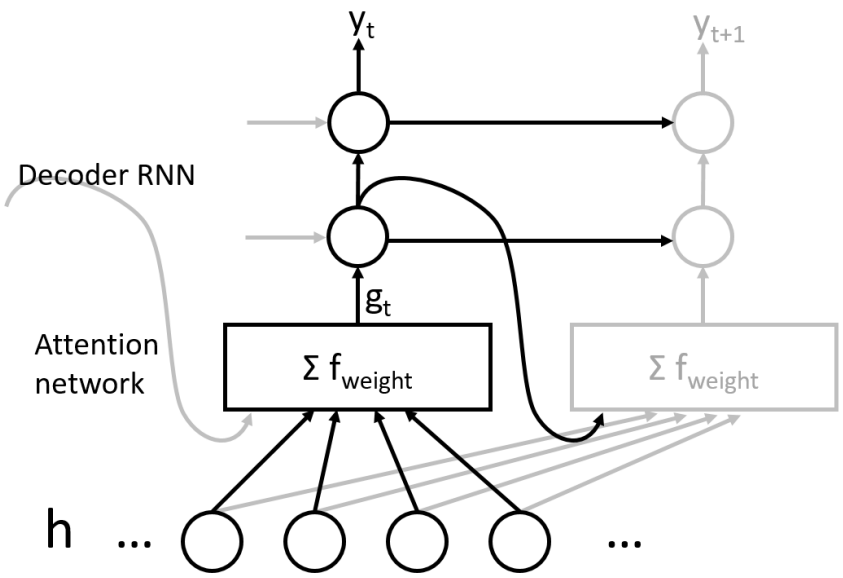

Figure 3-4: An attention-based model for translating from one sequence to another. This model generates the output sequence $y_{1:m}$ one element at a time. Each y_t is generated from a "glimpse" of the input sequence $h_{1:n}$, made from a linear combination of the h elements. The decoder RNN has control over the next glimpse, as well as the output y_t.

this problem is solved using an end token, which is added to the end of each training $y_{1:m}$. When the decoder generates an end token, decoding is stopped and the output sequence is considered finished.

3.5 Turning neural models into WFSTs

Neural network acoustic models can also be converted into WFSTs, so they can be combined with pronunciation and language models. An LSTM model trained using CTC loss can be represented as a WFST that emits phones at every timestep. The WFST has one state for every timestep and phone combination. At each state, it transitions to some phone at the next timestep, with probability equal to the LSTM output for that phone and timestep combination. However, the probability of remaining on the same phone is equal to the LSTM output for the current phone, plus the LSTM output for the empty frame. This reflects the CTC loss, which allows empty frames to stand in for any phone.

This WFST emits phone sequences with scores equal to $\log P(p_{1:n}|s_{1:n})$. As mentioned in Section 3.2, this probability can be converted into the likelihood $\log P(s_{1:n}|p_{1:n})$ by subtracting out the prior over phones $\log P(p_{1:n})$. Concretely, every score in the WFST can be decreased by the empirical probability in the training data of the phone emitted.

It is not feasible to exactly convert the output of an attention-based acoustic model into an WFST, because of the non-Markovian nature of the attention decoder: the i-th phone depends on all of $p_{1:i-1}$,

not just on some state at timestep $i-1$. The resulting WFST would have an exponential number of states. One solution is to make a WFST that represents only the a most likely phone sequences, which can be computed using beam search. Such a WFST would branch immediately at the starting state, with a different branches that are each a non-branching chain. However, a list of phone sequences represents far fewer possible decodings than a full lattice produced from an HMM, because the list does not have any combinatorial structure. In general, converting LSTM sequence models into lattices is an unsolved problem, and an issue that applies to neural language models as well.

Chapter 4

Speech synthesis using neural networks

The speech synthesis system discussed in this section applies the core ideas of speech modeling, which have been successful in recognition, to the inverse problem of synthesis. In this chapter, I show that neural network acoustic models can also be used to synthesize speech, in a configuration that is almost identical to how they are used for recognition.

4.1 Background and prior work

There are several different approaches to speech synthesis, as summarized in [22]. The physics of the human vocal tract can be modeled, and the model can be used to simulate waveforms generated for a particular set of phones. This approach, called formant synthesis or articulatory synthesis, requires manual model design, and is not amenable to automatic training using machine learning. Alternately, short samples of human speech can be recorded, and stitched together on demand to create the desired phones. This approach is called concatenative synthesis or unit selection synthesis, depending on the size of the sample bank.

All of these speech synthesis techniques require a high degree of manual intervention: the physical simulation techniques require linguists to design a phone model for each new language, and the concatenative techniques require a large speech corpus that has been annotated at the frame level.

In contrast, speech synthesis techniques based on machine learning do not require frame-level annotation. Traditionally, machine learning speech synthesis (often called parametric synthesis) has been dominated by HMM systems. The GMM-HMM acoustic model described in Section 2.2 is a generative model that jointly

models speech frames and phone labels. Therefore, it is possible to use the model in reverse, and extract the most probable speech frames, given some phone labels (or general HMM states) to synthesize.

In terms of the speech recognition framework discussed in Section 2, a speech synthesis system consists of an acoustic model and a pronunciation model. However, both models are used "backwards" - the pronunciation model converts words into per-frame phone labels, and the acoustic model converts per-frame phone labels into speech frames. In the probabilistic framework, a speech synthesizer models $\text{argmax}_S P(S|W)$, the most likely series of speech frames given some sentence. The distribution can be rewritten in terms of an acoustic model and a pronunciation model:

$$P(S|W) = \sum_{\text{all } P} P(S|P) \cdot P(P|W) \tag{4.1}$$

However, representing multiple phone and speech frame possibilities is less important in synthesis than it is in recognition. In a recognition problem, multiple frame-level phone alignments P can contribute to the same sentence W. Therefore, it is important to consider the sum over all possible phone alignments P, and assign an explicit likelihood to each one. In a synthesis problem, in contrast, multiple phone alignments P are unlikely to result in the same speech frames S with any substantial probability. The space of speech frames is much larger than the space of sentences. Therefore, synthesis systems usually produce the most likely phone sequence in a deterministic manner, followed by the most likely speech frames for that phone sequence:

$$P^* = \underset{P}{\text{argmax}}\, P(P|W) \tag{4.2}$$

$$\underset{S}{\text{argmax}}\, P(S|W) \approx \underset{S}{\text{argmax}}\, P(S|P^*) \tag{4.3}$$

In this non-probabilistic framework, both the pronunciation model and the acoustic model can be purely deterministic, and represent only the most likely output.

A speech synthesis pronunciation model relates words to per-frame phone labels. Like in a recognition pronunciation model, a pronunciation dictionary is used to get the phones for each word. Each phone must then be assigned a duration - the dictionary does not know how many frames each phone in a word should last. This *phone duration model* can simply be a table mapping phone identity (or triphone identity) to the average length of that phone in the training set. Average lengths can be extracted by training a recognizer HMM on the training set, then force-aligning the training set using the recognizer, and counting the lengths of each phone. The discounting techniques mentioned in Section 2.4 can be applied here, especially if triphone durations need to be generated. For example, if there is not enough data to estimate $\mathbf{E}[len(p_i)|p_{i-1} = a, p_i =$

$b, p_{i+1} = c]$, then some linear combination of the bigram expected length $\mathbf{E}[len(p_i)|p_{i-1} = a, p_i = b]$ and the unigram expected length can be used instead. RNNs can also be used to predict phone length [23, 24], by learning a mapping from a sequence of phones to a sequence of lengths for each phone.

The acoustic model in a synthesis system has traditionally been a GMM-HMM. There are, however, several differences between a GMM-HMM model used for synthesis and one used for recognition. The first is in the representation of speech frames. The 13-dimensional MFCC representation is commonly used for recognition. However, the MFCC transform does not preserve pitch, which is an important dimension of realistic speech. In the most basic solution to this problem, two additional features are extracted per speech frame: an average fundamental frequency (f_0), and a probability of voicing. (See [25] for a pitch extraction algorithm, and Chapter 3 of [26] for a description of how to reconstruct a raw speech waveform from MFCC features.) Higher dimensional feature vectors for speech synthesis, like STRAIGHT [27], contain a finer-grained MFCC histogram, delta and double delta (discrete derivative) values of the pitch and histogram, and other augmentations.

Compared to recognition HMMs, synthesis HMMs also have more complex phone representations. As mentioned in Section 2.2, recognition GMM-HMM systems typically use a triphone representation for each frame. To generate realistic speech, synthesis HMMs represent the phone at each frame with additional dimensions, including part of speech, accent, and type of sentence [26]. As a result, there will not be enough training data to cover every possible phone-plus-context value. To solve this problem, phones and contexts are usually clustered in a decision tree, in which p values that result in similar speech features are merged into a single group [28]. To construct a context decision tree, the training data is first force-aligned by training a context-free GMM-HMM model. Then, a list of (frame context, speech feature at that frame) tuples are created from the aligned training data. Every frame is initally put in a single cluster. Clusters are iteratively split by finding the context variable that splits the cluster into two clusters that have minimum entropy. This process is repeated until an entropy threshold is reached; then a context-dependent GMM-HMM is trained using the cluster labels as HMM states.

During synthesis, the GMM-HMM model will abruptly switch between Gaussian mixtures whenever a new phone is desired. This can result in audible discontinuities in the synthesized speech, if no post-processing is performed. To create smooth transitions between phones, the GMM can model the delta and double-delta values of each histogram value. During synthesis, the delta and double-delta values can be made consistent with the histogram values across neighboring frames [29]. This allows the GMM for each phone to enforce smoothness constraints on its neighbors, using its delta and double-delta values.

Recently, neural networks have been applied to the speech synthesis problem, demonstrating some improvement over HMM methods. In [30], a multi-layer dense neural network was used to learn a mapping

from context-dependent phone labels to per-frame speech features. Human volunteers subjectively judged speech generated by this network to be better than speech generated by an HMM. In [31], a stacked bidirectional LSTM was used to learn a mapping from a per-frame sequence of context-dependent phone labels to a sequence of speech features. In both of these systems, the training data consisted of clean, phonetically rich utterances recorded specifically for speech synthesis purposes (around 5 hours of speech each).

In this thesis, I remedy two weaknesses in prior work on neural network speech synthesis: the use of context-dependent phones, and the use of professional-quality training data. LSTM architectures are designed to learn dependencies across relatively long timescales, and to learn complex transformations between input and output. Therefore, context-dependent phone labels should not be necessary in an LSTM network. (In fact, some speech recognition LSTM systems [32] can even output graphemes (e.g. English alphabet characters) instead of phones.) Context-dependent phones require a context generator at synthesis time - each phone needs to be labeled with part-of-speech and articulation properties. Therefore, a context-free synthesizer will require a less complex pronunciation model.

Likewise, neural networks are specialized for learning patterns from very noisy training data. Recognition networks have had success on the switchboard corpus, which contains telephone-bandwidth recordings of natural conversation [33]. Thus far, synthesis networks have only been trained on purposely-recorded, single-speaker datasets. Experience with noisy datasets in other domains suggests that synthesis networks should be able to learn from real-world speech samples (e.g. captioned YouTube videos). This would make the creation of synthesized versions of arbitrary voices easier and less resource-intensive, and drastically expand the amount of training data available for training synthesis systems.

4.2 System design

Figure 4-1 shows all of the components of the synthesis model described in this section. The pronunciation model is made of a dictionary (identical to the one used in recognition models), which deterministically converts words to phones; and a n-gram phone timing model, which adds timing to the phones. The acoustic model is an LSTM that converts the phone time series into MFCC features. Training starts with a recognition model, which is used to compute the timing for the phones in the training data (a process called forced alignment). The force-aligned phones are then used to make the phone timing n-grams and LSTM synthesis models.

The phone timing model assumes a trigram distribution over the lengths of each phone. We build a table of $\mathbf{E}[len(p_i)|p_{i-1}, p_i, p_{i+1}]$, the expected length in speech frames of each phone, given the identity of the phone and the phones immediately before and after it. (The start and end of an utterance are defined using

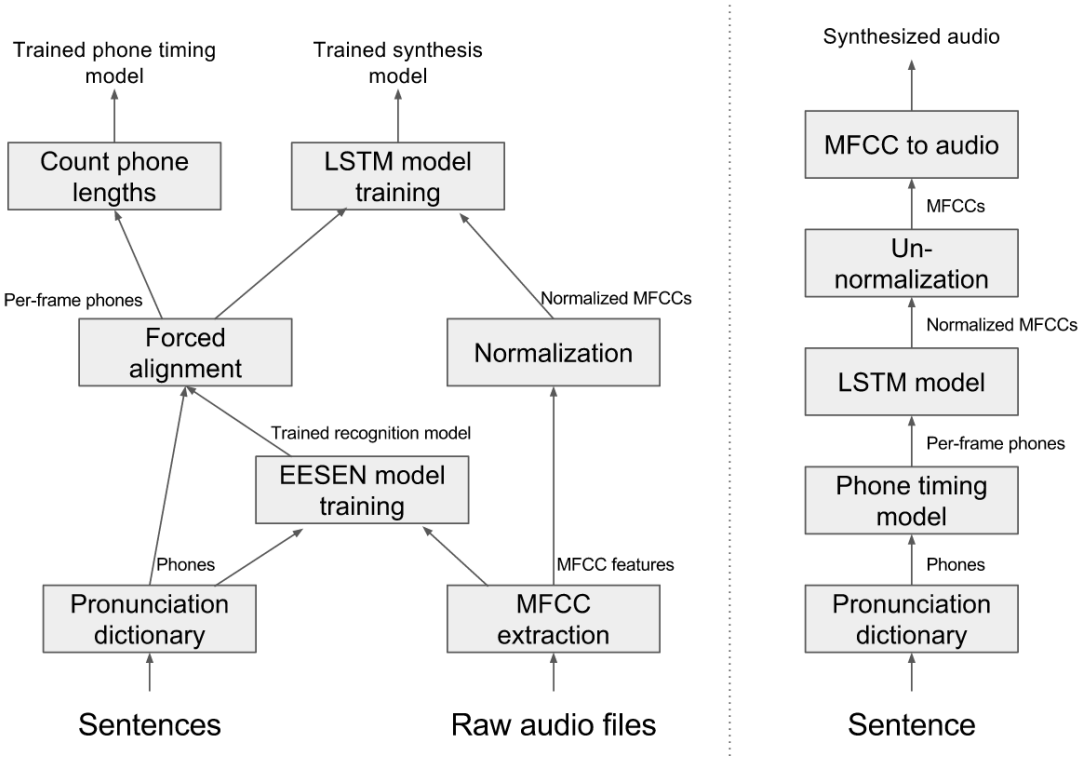

Figure 4-1: An overview of the components of the speech synthesis system described in this chapter, as they are used during training (left) and generation (right).

their own phone symbols.) This table is built from the training data in the following way: First, an EESEN attention-based speech recognizer [32] is trained on the training speech corpus. Then, the recognizer is used to force-align the training corpus, so that each frame has a phone label. We count the length of each phone by consolidating consecutive frames that have the same phone label. With the phone length counts, we can build a table of the average length of each phone, separated by phone context. If a particular triphone is not represented in the training data, we fall back to a diphone model. (More sophisticated interpolation techniques could be used here, as well.)

The acoustic model accepts as input a sequence of phones $p_{1:n}$, one for each timestep. Each phone is first converted into a n_p dimensional phone vector using a lookup table. Then, each phone vector is fed into a bidirectional LSTM: two separate LSTMs accept the phone vector sequence as input, one with recurrent connections forward in time and one backwards. Their outputs, of size n_l, are merged at each timestep. To encourage the network to learn more robust representations, we apply dropout [34] to the merged output: during training, half of the values in the output are randomly selected and set to 0. These zeroed values force the network to redundantly encode all operations, and discourage overfitting by changing the combination of weights used at each training iteration. More layers of bidirectional LSTMs (with dropout in between) follow. Finally, the output of the last LSTM is converted into a speech feature vector, by a single dense

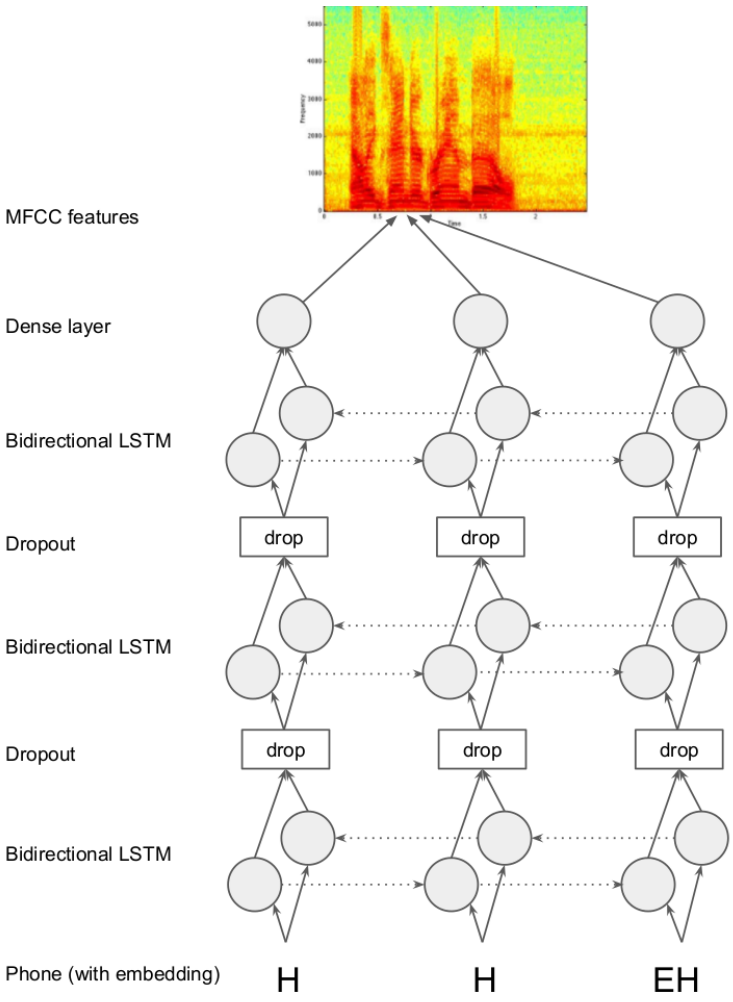

Figure 4-2: The LSTM-based speech synthesis neural network used in this section. The input is an embedding vector representing a single phone at each timestep, and the output is normalized MFCC features.

layer. This model is shown in Figure 4-2.

The acoustic model is designed to closely mimic the bidirectional LSTM plus dropout design that has been successful in speech recognition. Unlike almost all other synthesis models, it is defined in terms of context-independent phone labels, and relies on the LSTM layers to learn context. The dropout layers are intended to improve learning from very noisy training data. It is trained on the same force-aligned training corpus used in making the pronunciation model.

4.3 Training

The LSTM speech synthesis model was trained on two datasets: the TED-LIUM dataset [35], and Walter Lewin's lectures for MIT's electricity and magnetism course (8.02, Fall 2002). The TED-LIUM dataset

consists of 118 hours of speech, split into small talks of around 7 minutes each. Each talk is by a different speaker, although some speakers are represented multiple times. The speech is recorded from auditorium microphones, so it is generally of good quality, but not designed specifically for speech applications. The Lewin dataset consists of 36 lectures by the same speaker, each around 50 minutes long. The speech quality is noticeably poor in places: because the dataset was recorded from a live lecture, chalkboard scratching, audience noise, and audience questions are all common.

First, the EESEN recognizer was trained on the TED-LIUM dataset, using the default settings for phone recognition. Training was performed for 16 full iterations on the dataset, with a final cross-validation phone error rate of 18.91%. Because the TED-LIUM dataset was of higher quality than the Lewin dataset, this recognizer was used for all future experiments. The training dataset was then forced-aligned using Viterbi decoding on the EESEN model's posterior distributions over phones.

The speech files were then pre-processed in preparation for training the LSTM model. MFCC features, voicing probability, and f0 (fundamental frequency) were extracted from the audio files at 25ms intervals, using the Kaldi toolkit [36]. The 15-dimensional feature vector was then normalized so that each dimension has a mean of 0 and a standard deviation of 1 across the dataset. Normalization was done for each speaker in the TED-LIUM dataset, and across the entire corpus for the Lewin dataset. This normalization was primarily useful for measuring the error during training: a model that ignores the input phones cannot have a mean square error of less than 1 when the dataset is normalized. The extent to which the validation error was less than 1 provided a standardized measurement of how well the model was learning. For the TED-LIUM dataset, normalization also removes some inter-speaker differences, and allows us to emulate different speakers by adding in the mean and variance for a particular speaker at generation time.

The LSTM model itself was implemented using Keras on top of the Theano [37] GPU framework. The initial phone embedding was of size 64; the forward and backward LSTMs contained the same number of units, which was a hyperparameter we optimized. The model was trained using rmsprop, with the objective being the minimization of the mean squared error between the output of the dense layer and the expected speech features. A batch size of 16 was used: all of the utterances in the training set were sorted by length, and consecutive chunks of 16 utterances were loaded at a time into the model, with length differences resolved using zero padding. The gradient was averaged across the batch. Feeding the training data in batches accelerates the training process, and also appears to improve the stability of gradient descent. Table 4.1 shows the validation error for various configurations of the LSTM model.

On a single Nvidia Titan GPU, one iteration over the 100 hour TED-LIUM corpus takes about 4 hours for a two-layer model, and 8 hours for a three-layer model. The models in Table 4.1 were trained until convergence, which took between 10 and 20 iterations over the corpus. Table 4.2 shows the error rates on

each of the models after only two iterations. The different configurations converge at roughly the same rate per epoch; however, each epoch requires more time with more layers.

		Number of layers		
		1	2	3
LSTM size	128	0.770	0.697	0.683
	256	0.784	0.699	0.686
	512	0.770	0686	0.693

Table 4.1: Validation error for the LSTM model, trained until convergence.

		Number of layers		
		1	2	3
LSTM size	128	0.803	0.740	0.728
	256	0.813	0.728	0.723
	512	0.802	0.724	0.725

Table 4.2: Validation error for the LSTM model, trained for two iterations over the training data.

To generate speech for a given sentence using the trained model, the sentence is first converted into a list of phones, using the CMU Sphinx pronunciation dictionary. The phones are then given durations, using the pronunciation model. Then, the time series of phones is fed into the LSTM model, to make speech features. Finally, the speech features are converted into audio, using the RASTAMAT library [38]. The RASTAMAT library does not support pitch features by default - pitch reconstruction was added, using FM synthesis to make pitched noise.

4.4 Results

To generate the results below, we used a two-layer network with 512 units in each forward and backward LSTM, because the validation error of this configuration was as good as any of the three-layer configurations, and adding a third layer significantly increased the training time. Figure 4-3 shows a spectrogram of the result of synthesizing a sentence from a Wikipedia article. The synthesized audio is available for download, along with other samples from the synthesizer [1].

In general, the synthesized speech is very intelligable, although somewhat robotic. Some of the unnaturalness is due to a poor pitch synthesizer, which creates pitched noise that is too buzzy. (There is no open-source program for converting pitched MFCC features into sound files, so this part of the system was written from scratch.)

Because the mean and variance of each speaker's MFCC features were computed at training time, it is possible to re-adapt synthesized speech, by adding in the mean and multiplying by the variance of the

[1] `https://www.dropbox.com/sh/vqis4cjl7odjby9/AAB8u1umsxMaQS8xT7I14_k5a?dl=0`

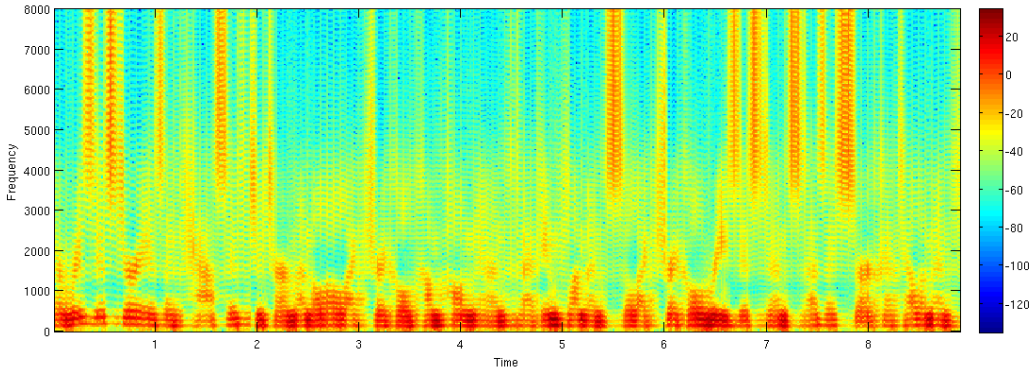

Figure 4-3: A spectrogram of speech synthesized by the system described in this section. The sentence was: "A resistor is a passive two terminal electrical component that implements electrical resistance as a circuit element".

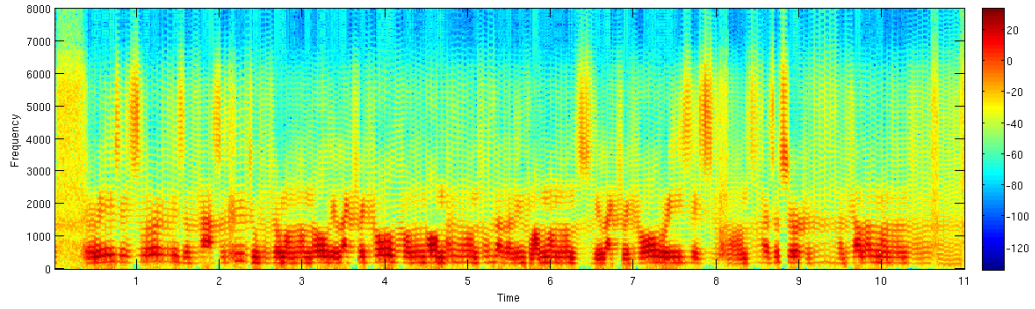

Figure 4-4: A spectrogram of speech synthesized by the system trained on the more challenging Walter Lewin dataset. The input sentence is the same as in Figure 4-3.

target speaker. This is demonstrated by the three "TED_Hello" clips in the samples folder. This process most noticeably affects the pitch of the sample, when switching between male and female speakers. It also changes some more subtle intonation properties. However, other speaker-specific properties, like stress pattern and rate of speech, cannot be adapted in this way, because they affect more than just the mean and variance of each feature.

In Figure 4-4, the synthesizer trained on Walter Lewin lecture data is asked to synthesize the same sentence. (This sample is also available for dowload in the Dropbox link.) Due to memory constraints and the longer average sentence length of the Lewin lectures, a network of two layers with 256 units was used. The final validation error was 0.831. The result is of lower quality than the speech from the TED synthesizer, likely because the Lewin training data is much noisier and somewhat shorter (30 hours vs. 100 hours). The speech is intelligable, but only with some effort. The prosody of the speech (the pacing of each syllable) is somewhat unnatural, because there are long pauses in the training data while Lewin is writing

on the chalkboard, which are reflected in the phone timing model. The result is that some phone n-grams are assigned unnaturally long durations. This problem could be fixed by detecting silence during the initial forced alignment of the speech corpus.

4.5 Conclusions

Using acoustic modeling ideas from the speech recognition community, we built a speech synthesizer that can learn directly from YouTube videos. This synthesizer tolerates noise in the training data at a level unseen in prior work: it learns from natural speech recorded in an ambient setting, and can aggregate data from multiple speakers. Compared to previous neural speech synthesis systems, it has a simpler architecture, that uses context-independent phones and only two LSTM layers.

4.6 Alternate approach - speech synthesis by inverting neural networks

In the previous sections, a speech synthesis acoustic model was described that mirrored the design of speech recognition acoustic models. The model was trained specifically for the synthesis task. In an alternate approach, a pre-trained speech recognition network can be inverted, without any additional training. To synthesize speech corresponding to a sentence, the system would search for an input to the recognition neural network whose output matches the sentence. In this section, a synthesis system using this principle is documented. Built on top of the EESEN recognizer, it takes a time series of phones (produced using the pronunciation model described in Section 4.2, for example) and produces a series of MFCC features. We find that the resulting sounds are in general not intelligible, although they contain speech-like features.

4.6.1 Neural network inversion

Given a trained neural network that maps inputs X to outputs Y, network inversion is a technique for sampling possible x values that would result in a particular given output y. Network inversion was introduced in the field of computer vision [39], where the technique can generate images that an object recognition network considers typical for a category.

Network inversion is essentially gradient descent over the input vector of a neural network. If a given

neural network f maps inputs to outputs as $y = f(x)$, then inverting the network is the same as solving for

$$x^* = \underset{x}{\operatorname{argmin}} |y - f(x)| \tag{4.4}$$

for some properly defined distance metric. Because f, being a neural network, must be fully differentiable, x^* can be approximated using gradient descent. Starting with a random guess for x_0^*, x^* can be updated using

$$x_i^* = x_{i-1}^* - \lambda \left(\frac{\partial}{\partial x} |y - f(x)| \right) \Bigg|_{x=x_{i-1}^*} \tag{4.5}$$

for some learning rate λ. More advanced optimization techniques, like momentum and adaptive gradient descent, can also be applied to this problem.

4.6.2 Experiment and results

The network inversion speech synthesizer was implemented on top of the Eesen system trained on TED-LIUM data as described in Section 4.3. The top-level C++ code that runs gradient descent over the training data was modified to compute the gradient of the input with respect to the classification error, as follows: A fake input layer was inserted into the network, before the first layer, with parameters that correspond to the elements of the input. All other parameters of the network were fixed in the backpropagation algorithm. The resulting modified network performs gradient descent over the input speech features, attempting to maximize the likelihood that the speech features decode to a given phone sequence.

When using network inversion to minimize a CTC error, the gradient descent algorithm will make the first frame of the speech input match the first phone of the desired output. Having done this, the fastest way to further reduce the CTC error is to make the second frame of the speech input match the second frame of the desired output. This way, there are two correct phones, instead of just one. However, this results in speech that is extremely rushed and unintelligible; we expect to hear several more frames of the first phone. The end result of gradient descent on a CTC loss function is speech that is packed into as few frames as possible, because this is the fastest way for a gradient descent algorithm to reduce the CTC loss to 0.

Instead, the CTC loss function was removed, and replaced with a cross entropy loss function for each frame of speech. The pronunciation model described in Section 4.2 was used to convert a target sentence into a time series of phones. Network inversion was used to minimize the cross entropy between the output of the EESEN network and the target phone series. 2000 iterations of gradient descent were used to generate the output shown in Figure 4-5. The result is only intelligable over the first three syllables ("this is a"), and sounds like garbled, but vaguely human, whispering afterwards. The spectrogram shows many speech-like

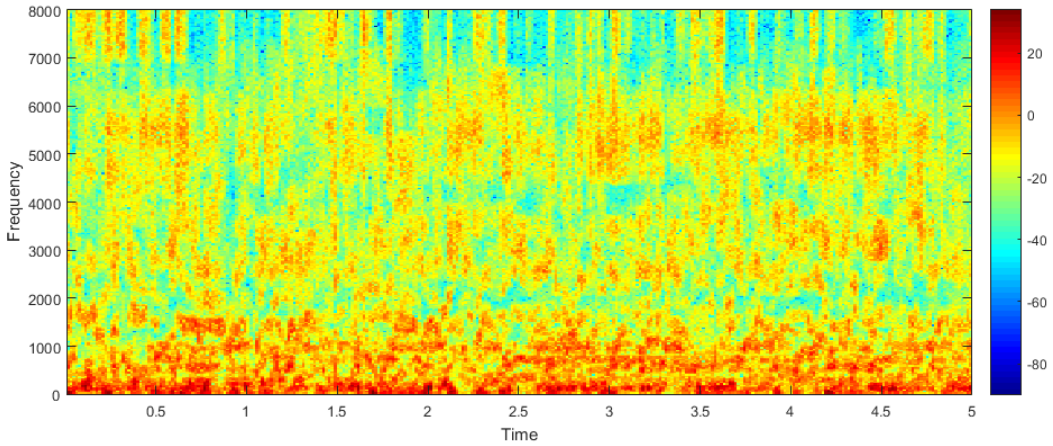

Figure 4-5: Spectrogram of "This is a relatively short test of my network", synthesized by inverting the Eesen speech recognition network.

features.

The main disadvantage of synthesizing speech using network inversion is that the synthesis process is very slow. To create MFCC features for a sentence, a thousand backpropagation steps may be needed, compared to one forward pass in a network trained for speech synthesis. As such, network inversion is unlikely to result in practical speech synthesizers. However, the technique may be useful for visualizing what a recognition network has learned.

Chapter 5

Multimodal speech recognition with images

In this section, I show how the speech modeling framework discussed in Section 2 can be adapted to the problem of multimodal speech recognition. I present results on a novel system that integrates speech and image cues for the same sentence, trained on the first multimodal recognition dataset of its kind. I show that image data can improve the accuracy of modern speech recognizers.

5.1 Prior work

There have been several research thrusts that attempt to model images using natural language. In the *caption scoring* problem, the most appropriate caption for an image must be chosen from a list of sentences. The problem is usually formulated as finding a function $f(image, caption)$ that returns a similarity score between the image and the caption. At decoding time, each proposed caption is scored against the test image, and the best-scoring caption is chosen.

Caption scoring neural networks typically use a convolutional neural network (CNN) to transform the image into a feature vector, which can be compared to another feature vector generated from the proposed caption using an RNN. The core unit of a convolutional neural network is a trainable convolution filter, which converts two-dimensional array (like an image) into another two-dimensional array. This convolution is defined as follows:

$$I_2(x,y) = \sigma\left(b + \sum_{i=-\delta}^{\delta}\sum_{j=-\delta}^{\delta} W(i,j)\cdot I_1(x+i, y+j)\right) \tag{5.1}$$

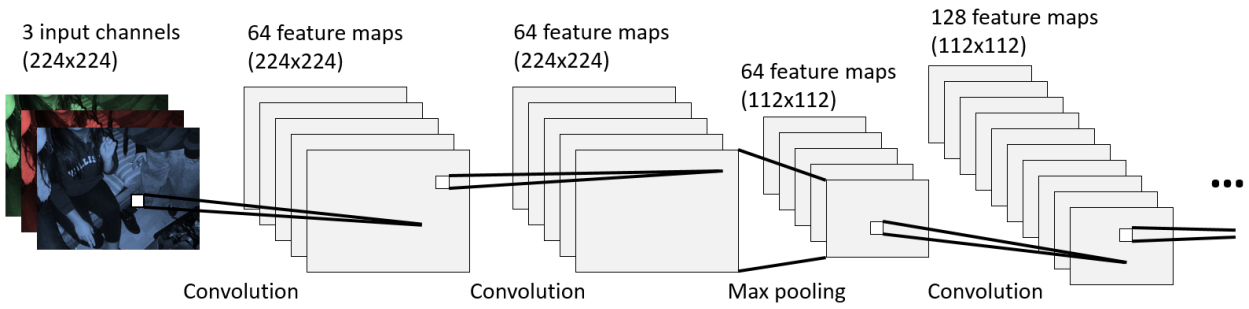

Figure 5-1: The initial layers of the VGG-16 convolutional neural network. There are 13 convolutional layers in the entire network, plus 3 fully connected layers at the end. The convolutions extract successively higher-level features from the input image, which the fully-connected layers use to decide on the identity of the image.

Here, the trainable parameter W is a $2\delta + 1$ square matrix that defines the kernel of the convolution, and b is a trainable scalar offset. As before, σ is a non-linearity, such as the *tanh* function. Edges of the image are typically padded with zeros, to allow the convolution to preserve image size. This trainable convolution is ideal for image processing, because it can identify simple features in an image, like edges. A stack of multiple convolution layers can identify more complex features, like circles and corners at a particular angle.

A CNN often contains *pooling layers* in between convolution layers. These layers reduce the size of the image, allowing higher-up layers to effectively "see" larger portions of the original image. The popular max-pooling layer takes the maximum value in a 2x2 block of pixels, reducing the image size by a factor of 2. Average pooling, in which the average of each 2x2 block of pixels is propagated, is also possible.

A CNN consists of multiple convolutional layers, sandwiched between pooling layers, with some dense layers at the top to create a one-dimensional output vector (which might represent a probability distribution over the item in the image, for example). Described in Figure 5-1 is the VGG-16 network [40], an object recognition network that is also used to extract image representations in many of the papers described below. In the VGG-16 network, multiple feature maps are created by applying convolutions with different kernels on the input image. Each new layer applies a convolution to one of the feature maps in the previous layer. In other CNNs [41], three-dimensional convolutions may be used instead, which combine several feature maps from the previous layer.

Recent approaches to caption scoring tend to use CNNs to extract features from images. In [42], the image is converted into a feature vector using a CNN. A separate CNN converts a bag-of-words representation of the sentence into a sentence feature vector. The final score between the image and the sentence is computed using a cross-correlation between the two feature vectors. Training attempts to maximize the correlation

between matching images and sentences, and minimize the correlation between non-matching pairs. The entire system (both CNNs plus the cross-correlation layer) is trained as one function, using gradient descent. In [43], a similar approach is used, except with a syntax tree-based neural network instead of a CNN to process the sentence.

In [44], regions of interest are extracted from each image using an object recognition bounding box algorithm. Then, a CNN representation is created for each region. Meanwhile, the sentence is fed through a LSTM network to create word vectors. Each word vector of the sentence is multiplied with each region vector of the image to create a similarity score between the word and the region. The total alignment score is defined in terms of the maximum similarity score for each word.

In the related *image captioning* problem, the goal is to generate a freeform caption for a given image. Early image captioning systems tended to paste together captions for similar images from a large database. The system described in [45] parses each sentence in the database into nouns, verbs, and prepositional phrases. It then extracts features from the images for matching each part of speech. For example, local SIFT features are used to match nouns, but global scene descriptors are used to match prepositional phrases. For a new image, it generates a bag of possible caption words, by picking words from the closest images in the database for each part of speech. An integer linear programming algorithm is used to order these words into a sentence.

More recent approaches use a recurrent neural network (RNN) to generate words of a sentence in order, from scratch. Vinyals et al. [46] fed the output of a CNN directly into the first hidden state of a word-generating RNN. The main disadvantage of this approach is that a fixed-size representation of the image is used to generate the caption, regardless of the complexity of the image. Xu et al. [20] attempt to fix this problem using an attention-based decoder, in which each word is generated from a dynamically-chosen sub-sample of the image.

5.2 System design

In the multimodal speech recognition problem, the goal is to transcribe an utterance that was spoken in the context of an image. Both the speech and the image are provided, as shown in Figure 5-2. Ideally, a multimodal system will use features from the image to be more accurate than an analogous acoustic-only system.

The probabilistic speech recognition framework can be modified to account for image data. There are now four variables: the spoken utterance S, the phone time-series P, the words W, and the image I. We are interested in the distribution over words conditioned on both the speech and the images, $P(W|S, I)$. We

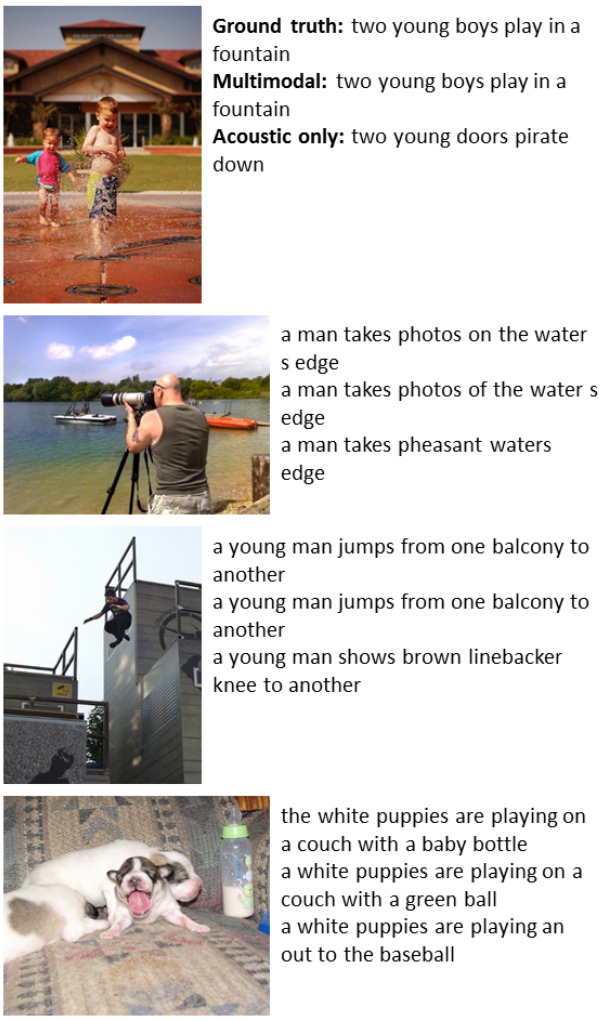

Figure 5-2: A demonstration of how image context informs speech decoding in our multimodal recognition system. The three sentences next to each image show the ground truth utterance, the decoding with image context ("multimodal"), and the decoding without image context ("acoustic only"). These examples were manually selected from the development set.

assume that P and S are independent of I given W; in other words, the image affects the speech only by affecting the words in the speech. The four variables form a Markov chain as $S \leftrightarrow P \leftrightarrow W \leftrightarrow I$. (One can imagine some realistic violations to this assumption, like a very exciting or scary image that may change the speaker's tone.) With this assumption, the desired distribution can be computed as follows:

$$P(W|S,I) = \frac{P(S,W|I)}{P(S)} \tag{5.2}$$

$$= \sum_{\text{all } P} \frac{P(S,P,W|I)}{P(S)} \tag{5.3}$$

$$= \sum_{\text{all } P} \frac{P(W|I) \cdot P(P|W,I) \cdot P(S|P,W,I)}{P(S)} \tag{5.4}$$

$$= \sum_{\text{all } P} \frac{P(W|I) \cdot P(P|W) \cdot P(S|P)}{P(S)} \tag{5.5}$$

$$\propto \sum_{\text{all } P} P(W|I) \cdot P(P|W) \cdot P(S|P) \tag{5.6}$$

Compared with Equation 2.6 (reproduced here),

$$P(W|S) \propto \sum_{P} P(S|P) \cdot P(P|W) \cdot P(W) \tag{2.6}$$

the only difference is that the language model $P(W)$ is now an image-conditioned language model $P(W|I)$. Therefore, this section will focus on the design of the new image captioning model. Standard Kaldi [36] tools, described in the Section 5.3, are used to build the acoustic and pronunciation models.

Our image captioning model is built on top of the "neuraltalk2" library [44], which encodes images using a CNN, and generates words from the encoded image using an LSTM. The overall network is summarized in Figure 5-3. The VGG-16 image network discussed in Section 5.1 is used to extract features from the image to be captioned. These features are used as the initial state of an LSTM model over words. At timestep i, the LSTM network takes as input a one-hot representation of the $i-1$-th word in the sentence, and produces as output a probability distribution over the i-th word in the sentence. This model can be used to score the match between an image and a caption, by multiplying the probability of each word in the caption, as defined by the output of the LSTM. It can also be used to generate a caption for an image, by sampling one word at a time from the output of the LSTM.

We define the image captioning model as the weighted combination of two components: a trigram language model P_{lm}, and a RNN caption-scoring model P_{rnn}. The total caption probability is

$$P(W|I) = P_{lm}(W|I)^{\alpha} \cdot P_{rnn}(W|I)^{1-\alpha} \tag{5.7}$$

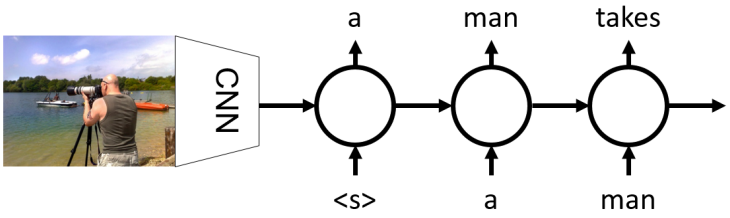

Figure 5-3: An overview of the neuraltalk2 architecture. The CNN converts each image into a vector, which is used by the LSTM to generate the caption, one word at a time. The output of the LSTM is a probability distribution over the vocabulary, but the diagram shows only the most likely word, for simplicity.

The trigram model is faster but less precise than the RNN model, and is used to prune the decoding lattice in a first pass. This language model approximates the true $P(W|I)$ by sampling many sentences from the caption generation model, and then summarizing the sentences in a trigram model. As such, it is specific to each image. A large number N_c of captions are generated for the image, using the neuraltalk2 model. These captions are combined with all of the real captions in the training set, and a trigram language model is trained on the entire combined corpus. The generated captions are intended to bias the language model towards words and short phrases that are more likely, given the image. The trigram model is not designed to be precise enough to reliably pick out only the correct sentence; rather, it is designed to preserve in the lattice a number of possible sentences that could be correct, so that the more precise RNN language model can then find the best one.

The resulting trigram model can be used in the Kaldi speech recognition toolkit [36], in place of a regular language model. From the resulting lattices, the 100 most likely sentences for each utterance are extracted, and rescored using the full $P(W|S, I)$: a weighted combination of the acoustic model, the image-conditioned trigram model, and the neuraltalk2 RNN caption scoring model. The most likely sentence at this point is returned as the final answer. The recognition process is summarized in Figure 5-4.

5.3 Training

5.3.1 Data

We train and evaluate our multimodal recognition system on the spoken Flickr8k dataset. The original Flickr8k dataset [47] consists of 8000 images of people or animals in action from the Flickr photo community website. Each image was described by 5 human volunteers, resulting in 40,000 descriptive sentences.

The spoken Flickr8k dataset, by Harwath and Glass [48], contains spoken recordings of all 40,000 sentences. The audio was collected via Amazon Mechanical Turk, an online marketplace for small human tasks. Volunteers ("Turkers") were asked to speak each caption from the Flickr8k dataset into their computer

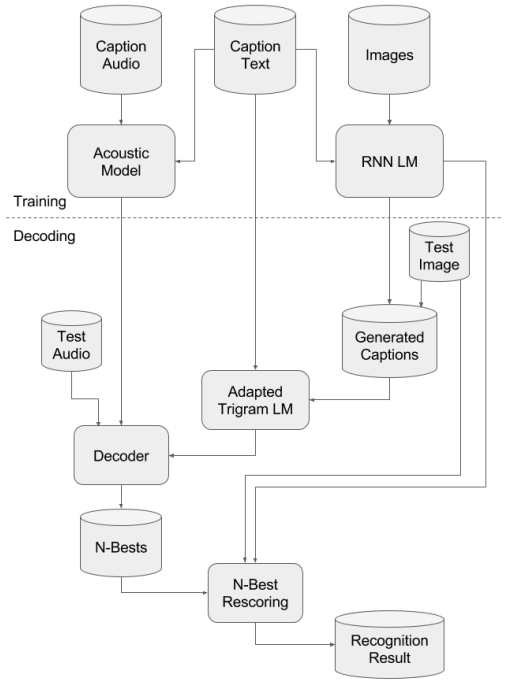

Figure 5-4: Configuration of the multimodal recognition system. Two models need to be trained: a Kaldi acoustic model and an image-captioning model ("RNN LM"). During decoding, the image-captioning model generates captions for the test image to build a better language model, and also rescores the top final hypotheses.

microphone. The speech was very roughly verified using Google's speech recognition service: an utterance was rejected if a majority of the words in the caption could not be detected. Turkers were paid 0.5 US cents per successful caption spoken. 183 Turkers participated in this task, recording an average of just over 200 sentences/person. Due to the distributed crowdsourced collection procedure, the quality of the recordings is highly variable. As such, this dataset represents a challenging open-ended speech recognition problem.

The dataset was partitioned into training, development, and test sets using the official published Flickr8k split. 6000 images (with 30,000 sentences in all) were assigned to the training set, and 1000 images (5000 sentences) to each of the development and test sets. Note that there is speaker overlap between the three splits: some utterances in the training and testing sets are spoken by the same speaker.

5.3.2 Baseline recognizer and acoustic model

We first train a Kaldi recognizer on the 30,000 spoken captions from the Flickr8k training set. Our baseline recognizer is trained using the default Kaldi recipe for the "WSJ tri2" configuration, which uses a 13-dimensional MFCC feature representation plus first and second derivatives. Feature are normalized so that each dimension has zero mean and unit variance. The acoustic model from this recognizer is also used to

model $P(S|P)$ for all of our multimodal experiments.

5.3.3 Building the trigram model

First, the image captioning model was trained on the 6,000 image and 30,000 caption training set, using the default parameters in the neuraltalk2 GitHub repository. The neuraltalk2 training process initializes parameters from a pre-trained VGG-16 network. This pre-trained network reduces the training time, and improves performance on the relatively small Flickr8k dataset. Stochastic gradient descent is then used to maximize the probability of each training caption, given the training image associated with the caption. The objective function used is:

$$L(I, w_{1:m}; \theta) = \prod_i P(w_i|w_{1:i-1}, I; \theta) \tag{5.8}$$

Each $P(w_i)$ is taken from the output layer of the LSTM at the i-th timestep. The i-th LSTM output depends on the previous words $w_{1:i-1}$, because the previous word is fed into the LSTM at each timestep. Because the neuraltalk2 model is differentiable from end-to-end, all of the parameters of the model (both the CNN and the LSTM) can be optimized together.

To make the trigram language model, we use the trained neuraltalk2 model to sample N_c captions for each image in the dev/test set, and append these captions to the existing training set captions to create a training corpus for each image. We then optimize the discounting parameters for a trigram model with Knesser-Ney interpolation (using the kaldi_lm library). The result is a different language model for each image.

There are two parameters to adjust for the trigram model: N_c, the number of generated captions to add to the model, and T, the "temperature" of the caption generator. T controls how randomly the caption generator behaves: Words are generated one at a time using the output distribution of the LSTM $P(w_i|w_{1:i-1}, I)$. The probability of possible words in this distribution can be rescaled by T:

$$P_{new}(w_i|w_{1:i-1}) = \frac{\exp(P(w_i|w_{1:i-1}, I)/T)}{\sum_w \exp(P(w|w_{1:i-1}, I)/T)} \tag{5.9}$$

The sampling process is more likely to pick the highest-scoring words at a low temperature, and more likely to pick random words at a high temperature. The optimal temperature must strike a balance between not generalizing to the testing data at the low end, and not providing any useful information about the image at the high end.

To tune these parameters, and to assess whether the addition of generated captions improves the performance of our language model, we measure the perplexity of our language model on the development set,

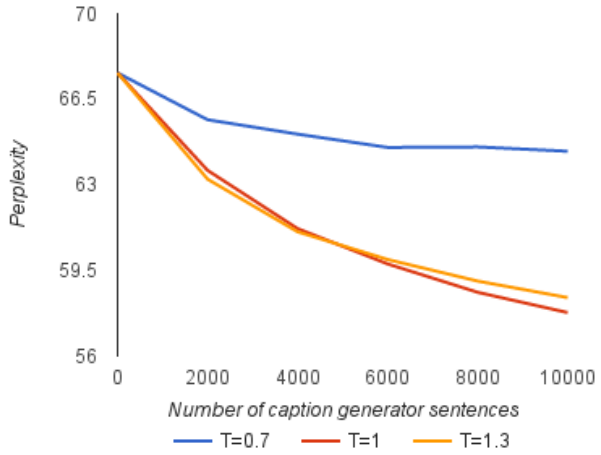

Figure 5-5: Perplexity on the development set, for various image-augmented language model parameters.

consisting of 1000 images with 5 captions each. The perplexity of a language model P_{lm} over a sentence $w_{1:m}$ is defined as:

$$perp(w_{1:m}; lm) = \frac{1}{m} \cdot \exp(-\sum_{i}^{m} P_{lm}(w_i) \log P_{lm}(w_i)) \tag{5.10}$$

This definition is analogous to the exponentiation of the entropy of the sentence given the language model, averaged over the number of words in the sentence. Because it is monotonically related to entropy, a lower perplexity implies a better language model.

The results are shown in Figure 5-5. In general, our image-augmented language model is significantly better at modeling the held-out development set than a standard trigram model trained on only the given captions. To verify that the better performance is not just due to more training data, we train a trigram model on the training corpus, plus 10 sentences chosen randomly from each of the 1000 development images. The perplexity of this trigram model was 66.06, which is essentially the same as the trigram model trained on only the training corpus. In our subsequent experiments, we used the best configuration found in cross-validation, which was 10000 captions with a temperature of 1.0.

Figure 5-5 suggests that increasing the number of captions beyond 10,000 could further improve the trigram model. However, the caption generation process, which must be performed at test-time for each image, is the bottleneck in the decoding process, taking several minutes per image for 10,000 captions. For practical applications, generating even more captions would be prohibitively slow. In the future, it may be possible to build a language model tree using the caption generator, by doing a breadth-first search over the most likely sentences, starting from the first word. The size of a tree language model is still linear in the number of hypotheses represented, but the generation process is more efficient.

5.3.4 Rescoring using the RNN model

Using the trained acoustic model and the image-augmented trigram model, we build decoding WFSTs using the standard Kaldi procedure. We construct a language model WFST out of the image-augmented trigram model for each image. We also construct one pronunciation WFST (called the "grammar WFST") for the entire system. The two WFSTs are composed and minimized using the OpenFST library, to create one WFST that relates phone sequences to words. We then build an acoustic WFST out of the GMM likelihoods for each frame of speech, and compose this with the existing WFST, to make the final decoding WFST.

The 100 most likely sentences for each test utterance are extracted from the final decoding WFST using Kaldi's `lattice-to-nbest` script. We rescore each sentence using the acoustic, trigram, and RNN models together. Grid search over the weights of each model was performed on the development set, to find the best linear combination of scores from the three models.

5.4 Results and analysis

Table 5.1 shows the word error rate (WER) of the multimodal recognition system in various configurations on the 5000 utterance development set. The full multimodal system decreases the WER by about 0.8 percentage points. Most of the WER improvement is due to the RNN rescoring of the top hypotheses; the image-augmented trigram model contributes only modestly.

5.4.1 Oracle experiments

Next, we perform some experiments to explore the performance of each of the components of the model in more detail. The trigram model was designed to ensure that the decoding lattice contains sentences with the correct key words and phrases. To measure the extent to which this is happening, we compute the accuracy of the *best* hypothesis in the top 100 hypotheses in each lattice. This is equivalent to assuming that the RNN model is perfect, and can pick out the most likely sentence, as long as that sentence is one of the choices.

System	WER (%)
Acoustic + LM *(baseline)*	15.27
Acoustic + LM + RNN	14.53
Acoustic + Image-LM	15.15
Acoustic + Image-LM + RNN	14.43

Table 5.1: Word error rates on the Flickr8k development set. LM refers to the trigram language model trained on only the training captions; Image-LM refers to the trigram model trained on the training captions plus the captions generated by neuraltalk to describe the image. RNN refers to the caption-scoring neuraltalk model.

(We therefore call this the *RNN oracle* score.) If the image-augmented trigram model is working correctly, it should result in a higher RNN oracle score than the standard language model.

In Table 5.2, we see that the RNN oracle score for the image-augmented trigram model is only modestly better than the RNN oracle score for the baseline trigram model. The difference between the RNN oracle WERs of the image LM and the baseline LM (0.11%) is about the same as the difference between the WERs of the Acoustic + Image-LM model and the Acoustic + LM baseline (0.12%). This suggests that image-augmented trigram model does not effectively "rescue" the correct sentence from being pruned out of the lattice. If the image trigram model were working as intended, it would pack the lattice with more probable sentences, making the difference higher when a perfect RNN is used to rescore the lattice.

We can also analyze the RNN model in isolation - even if the trigram model can reliably put the correct answer in the lattice, can the RNN model identify the correct answer? To do this, we add the ground truth sentence to the top 100 hypotheses from the lattice, and use the RNN model alone to rescore all 101 sentences. We compute the WER of the sentence that the RNN model marks as the best. We call this the *acoustic oracle* model, because this is equivalent to using a perfect acoustic (plus language) model always generates the correct hypothesis.

We would expect the acoustic oracle system with RNN-only rescoring to exhibit worse performance than the actual system, because the final rescoring step does not use any acoustic or language model information. The RNN model alone must pick the correct sentence from a list of 101. However, the results in Table 5.2 show that, when the correct sentence is a choice, the RNN model is very good at finding it. The acoustic oracle system is in fact more accurate than any of the full-rescoring systems. In contrast, if we rescore the top hypotheses without the correct sentence as a choice ("Default lattice, rescore with RNN only"), the error rate increases dramatically.

The acoustic oracle experiment suggests that this recognition system is weak in building decoding lattices that contain relevant sentences, not in choosing the best sentence out of the lattice. However, the RNN oracle experiment shows that the lattices already contain hypotheses that are much better than the ones chosen by

System	WER (%)
Acoustic + Image-LM + RNN	14.43
RNN oracle, baseline LM	8.55
RNN oracle, image-LM	8.44
Default lattice, rescore with RNN only	17.57
Acoustic oracle, rescore with RNN only	12.71

Table 5.2: Word error rates on the Flickr8k development set, for models with oracle components. In the RNN oracle model, we assume that the RNN model assigns a cost of 0 to the most correct sentence, and infinity to every other sentence. In the acoustic oracle model, we assume that the decoding lattice always contains the ground truth sentence.

the rescoring - the best sentence in the lattice has a 8.5 % WER on average, while the sentence chosen by rescoring has a 14.4 % WER on average. Between these two propositions, it may be that the RNN model is better at recognizing the exact sentence to describe an image, than it is at recognizing sentences that are slightly different. When the correct sentence is not present, the RNN model has trouble picking the best alternative that is in the lattice.

5.4.2 Test set

Finally, we present results on the Flickr8k test set. In light of the development set experiments showing that the RNN rescoring model provided most of the word error rate improvement, we also built speaker-adapted (SAT) versions of the Acoustic + LM and Acoustic + LM + RNN models. (In the Kaldi framework, it is difficult to perform speaker adaptation across multiple language models, so we did not apply SAT to the Image-LM models.)

To train speaker-adapted models, we first learn a transformation of the training data using linear discriminant analysis (LDA) [49]. LDA finds a linear operator that maximizes the difference between feature vectors from different speakers. We re-map the entire training set using the resulting transform before training the GMM-HMM. We also use maximum likelihood linear transforms (MLLT) [50] to adjust the GMM parameters, after initial GMM-HMM training. MLLT fits a per-speaker linear transform of the GMM parameters, to increase the likelihood of the speaker's training utterances. Both of these procedures are intended to reduce the burden on the speaker-independent acoustic model of modeling inter-speaker variation.

System	WER (%)
Acoustic + LM	14.75
Acoustic + Image-LM + RNN	13.81
Acoustic (SAT) + LM	11.64
Acoustic (SAT) + LM + RNN	11.08

Table 5.3: Word error rates on the Flickr8k test set. SAT refers to a speaker-adapted acoustic model.

Table 5.3 shows that adding image context improves the WER, both with and without speaker adaptation. The improvement is somewhat less in the speaker-adapted model, because the base accuracy is better and the image-augmented language model cannot be used.

5.4.3 Experiments with the Flickr30k dataset

Because the Flickr8k training set, with 6000 images, is relatively small for training image neural networks, we also experiment with training the neuraltalk2 model on the larger Flickr30k dataset [51], which consists of 31,783 images with 5 captions each, chosen to be similar in content to the Flickr8k dataset. We use all

of these images for training, and the original Flickr8k development and testing sets for development and testing. Therefore, our results here are directly comparable to our results on the Flickr8k dataset. Only the rescoring RNN and the caption generator use the Flickr30k dataset; the remaining components of the system (including the acoustic model) are trained as before using the Flickr8k dataset.

Table 5.4 shows that our system performs markedly better when trained with the Flickr30k dataset. In particular, the image-augmented language model alone, which was essentially ineffective when trained with the Flickr8k dataset, now improves recognition by over one percentage point from the baseline. Moreover, this accuracy improvement remains even when the RNN is used to rescore the top captions, showing that the image-LM and the RNN are now making orthogonal contributions. The full model now improves the WER by 2.8 percentage points.

System	WER (%)
Acoustic + LM	14.75
Acoustic + Image-LM + RNN (Flickr8k)	13.81
Acoustic + Image-LM (Flickr30k)	13.31
Acoustic + LM + RNN (Flickr30k)	13.20
Acoustic + Image-LM + RNN (Flickr30k)	11.95
Acoustic + LM + RNN-LM (No images)	13.79

Table 5.4: Middle: word error rates on the Flickr8k test set, using a model trained on the larger Flickr30k dataset. Botton: word error rate on the Flickr8k test set, using an RNN language model that does not use any image information, trained on the Flickr30k captions.

The larger accuracy improvements raise an important question: how much of the improvement is due to rescoring using an RNN language model, and how much is actually due to the image data? Our image captioning network models a distribution over word sequences using an RNN, which is inherently more powerful than an n-gram language model, because RNNs use a larger context when deciding on the probability of each word. To measure how much of the WER improvement is due to the RNN itself, as opposed to the image, we build an RNN language model that essentially mimics the RNN part of neuraltalk2, and use it to rescore the top 100 captions from the decoding lattice.

Our RNN language model is shown in Figure 5-6. It consists of a vector embedding for each word in the vocabulary, followed by one LSTM layer and one per-timestep linear layer. Softmax normalization is applied to the output of the dense layer, which is then interpreted as a probability distribution over the next word in the sentence. This is exactly the sentence generator used in neuraltalk2, down to the size of each layer. The only difference is the initialization of the hidden state at the first timestep of the LSTM, which is the zero vector in our RNN language model, but is the output of the CNN in neuraltalk2.

This language model was implemented using Keras and Theano, and trained on all of the sentences from the Flickr30k dataset. Any word that shows up fewer than 5 times in the training set is replaced with an

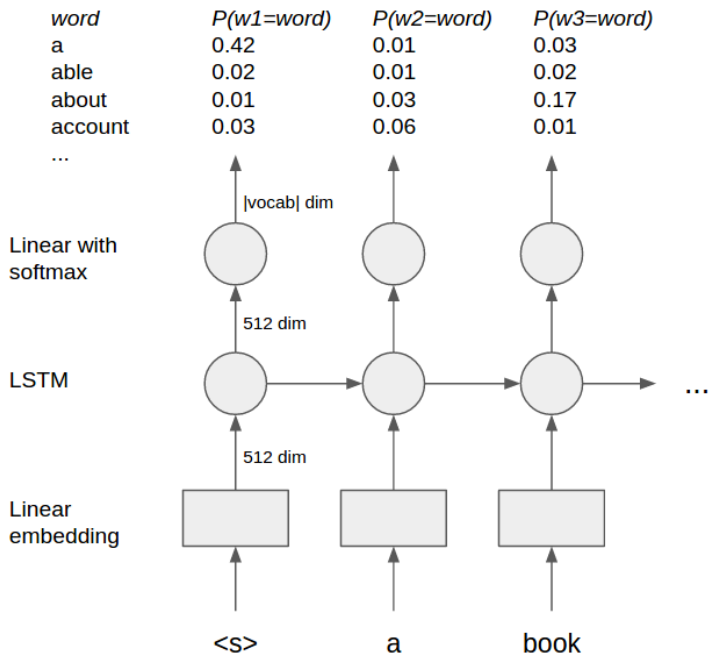

Figure 5-6: The RNN language model, based on the language model portion of neuraltalk2.

UNK (unknown word) token, just as in the neuraltalk2 training script. A small hold-out dataset was cut from the training set, to be used in monitoring the convergence of the training process. At the bottom of Table 5.4, we see that the RNN language model improves accuracy over the Kaldi baseline, but not as much as any of the models that use image data. We conclude that our multimodal recognition model is in fact effectively integrating image information.

5.5 Conclusions

If an utterance is spoken in the context of some image, we showed that the image can provide information that improves speech recognition. We found that taking the output of a conventional speech recognizer and rescoring the most likely sentences using a caption scoring model is the single most effective strategy for reducing word error rate. However, we found evidence that our caption scoring model was not good at identifying the closest sentence, when none of the most likely sentences were exactly correct. This may be a consequence of the way the caption model is trained: during training, the caption model is used to pick the right caption from a set of captions for randomly-selected images. It is never asked to discriminate among many captions for the same image. If this is the case, better results may be obtainable by training the caption scoring model in a way that is more similar to how we use it.

We also experimented with lattice-compatible image models, which can be integrated into a decoding lattice. Compared to rescoring the top hypotheses, a lattice-based approach can be used to explore a much wider range of possible decodings, because of the inherent efficiency of the lattice representation. We built a trigram language model based on captions generated from the image, which improved the accuracy of our recognizer by a small amount. Our image-augmented trigram model seemed to perform disproportionally better when the training dataset was expanded.

Our work shows that integrating image cues into speech recognizers is a promising approach, when appropriate visual data are available. The probabilistic speech recognition framework is easily amenable to additional sources of information.

Chapter 6

Conclusions

The overall theme of this thesis is that the speech models used in recognition can easily be adapted to solve other problems. We show two examples of novel systems based on speech recognition models: a speech synthesizer and a multimodal image and speech recognizer. The speech synthesizer uses a neural network to represent the relationship between phones and speech frames, following the example of speech recognition acoustic models. The multimodal recognizer integrates image data into a decoding lattice, by building a language model based on the test image and by rescoring top recognition hypotheses using a caption scoring RNN.

Both of these systems advance the state of the art in at least one dimension. The synthesizer is the first system (to our knowledge) that is trained entirely with noisy YouTube videos from multiple speakers. Previous work on synthesis has focused on purpose-recorded, single-speaker datasets. The multimodal recognizer is the first attempt at using image data as context for a speech recognition task. It improves recognition accuracy by almost 3 percentage points from an acoustic-only baseline, and scales well with the size of the image training data.

The work presented in this thesis offers possibilities for new applications, including the "cloning" of a person's voice from natural recordings and spoken language assistants that are aware of their surroundings. Moreover, this work demonstrates how speech recognition components can be disassembled and combined with other AI systems, so that even more systems can be built in the future.

Chapter 7

Appendix

7.1 Lessons learned - Software engineering for neural networks

The design of neural networks introduces new software engineering challenges. Most research has focused on making neural networks that are more powerful or accurate for some application; but the problem of making neural network *code* that is maintainable, easy to iterate on, and reliable is a challenge that all programmers working with neural networks must eventually face. Since relatively little has been written on the subject of software engineering for neural networks, I wish to use this section to share some informal insights gained in making this thesis.

In this section, I will focus on the idea of *caching*, which has many applications in neural network programming. Models parameters can be cached to save the state of a training procedure, models themselves can be cached to reduce the time it takes to initialize a neural network, and results can be cached to avoid using the neural network altogether. Properly used, caching can make designing neural networks much faster, while leaving code that is still transparent and easy to maintain.

7.1.1 Model caching

In the Theano framework, models must be compiled before they are used. During compilation, symbolic derivatives are computed for the variables in the network, and GPU code is generated for the operations that the network must support. This process can take over 10 minutes for a moderately-sized network, and must be performed each time the Python module is loaded. Model compiling can become the bottleneck to writing and debugging neural network code, especially if the programmer does not need to fully train the network before making the next adjustment to the code. During testing, when the trained network is used

```
m = Sequential()
m.add(Embedding(vocab_size, 512))
m.add(LSTM(512, return_sequences=True))
m.add(TimeDistributedDense(vocab_size))
m.add(Activation('softmax'))
overwrite = False
# m.compile(optimizer='rmsprop', loss='categorical_crossentropy')
m = model_cache.cache_compile(m, overwrite,
        optimizer='rmsprop', loss='categorical_crossentropy')
```

Figure 7-1: Example of how `model_cache` is used to compile a simple model. The standard Keras compile command, which is replaced by calling `model_cache`, is commented out.

to process data, model compiling almost always dominates the total processing time.

To avoid recompiling the same model, the compiled model can be serialized and saved into a model cache. If the same model is needed again, it can be loaded in pre-compiled form from the cache. The model cache will suffer a cache miss whenever the parameters of the model (the number of layers, etc.) change, because these parameters cannot be modified in the compiled model. However, I found that in the majority of the times I ran a Python module containing a Theano model, I was using a model that I had compiled at least once before.

To implement model caching in Keras, I built a `model_cache` library, which allows a Keras user to add model caching to any existing model with a one-line modification. The library exposes one function: `model_cache.cache_compile(model, **kwargs)`, which takes in an uncompiled Keras model, and returns a compiled Keras model. The cache compiler is designed to be a drop-in replacement for the `model.compile()` method call that Keras programmers normally use to compile models. Example usage is shown in Figure 7-1.

The model cache maintains a static JSON dictionary on disk, with a record of every model seen so far. The JSON dictionary maps a hashcode of the model to a file location where the model is cached. The cPickle library is used to directly save and load compiled Keras model objects, which contain within them all of the necessary GPU code. When `cache_compile` is called with a model, the model is hashed, and either the compiled model is fetched from disk using the hash; or in the case of a cache miss, the model is compiled and saved to disk, and the JSON dictionary is updated.

The crucial operation in the model cache is hashing the model. The model hash function must quickly convert a Keras model object into a 64-bit integer, such that different models have different hashes with very high probability. Our model hash function takes advantage of the fact that Keras builds a configuration dictionary as layers are added to the model, which includes an ordered list of the layers in the model. We add the compiliation keyword arguments to this dictionary (so that compiling with a different optimizer will

result in a different model hash), and hash the key-value pairs of the dictionary in sorted order. This process ensures that any two models with different configurations will hash to the same value with the hash collision probability of the hash function, which is extremely low.

In practice, I estimate that the model cache prevented several hundred redundant compilations, over the course of a few months. This has in turn saved several days of development time. As implemented, there are minor issues that make the cache more brittle than would be ideal. Major updates to Keras or Theano can break the configuration dictionary format, requiring the cache to be wiped. Likewise, custom layers are supported, but if their implementation changes, the cache will need to be manually invalidated.

7.1.2 Weight caching

The process of storing weights is relatively straightforward in Keras - the `model.save_weights` function dumps all of the weights of a model into a file, where they can be loaded into another instance of the same model using `model.load_weights`. There are a few practical things worth mentioning about weight saving.

First is that saving a model, as described in the previous section, also saves all of the weights associated with the model. However, saved models are an order of magnitude larger on disk than saved weights from the same model, and also take at least an order of magnitude longer to load. Therefore, model saving should only be used to speed up compilation, and weight saving for everything else.

Second is that model weights should be saved often during training. Often, models will overfit if left to train for too many iterations: the validation error will begin increasing. The decision of when to stop training a model becomes much less stressful if the model weights are saved every time validation is run. This way, the programmer can look through the validation log to find and load the best model so far, at any time in the training process. This also minimizes the amount of work lost if the training machine unexpectedly restarts.

7.1.3 Result caching

If a trained model is used many times to evaluate different test sets, caching the evaluation results may save a lot of time. This was the case for the grid search procedure in the multimodal recognizer - the entire test set had to be evaluated for around 100 different weight combinations. The bottleneck for this procedure is rescoring the top 100 recognition hypotheses using the neuraltalk2 network. The development set had 5000 utterances, each with 100 hypotheses. This means the network has to evaluate 50 million sentences across the grid search procedure. However, these sentences are not unique - changing the weighting of the acoustic and language models slightly will not change most of the 100 best hypotheses.

Therefore, I implemented a results cache for neuraltalk2, which saves the scores of all sentence-image combinations seen so far. The cache was initially implemented as a single JSON dictionary, with the sentence concatenated with the image filename as the key, and the score as the value. However, this did not allow multiple evaluation scripts to run in parallel. In order for a script to update the cache, it must read the JSON file, update it in memory, and write the JSON dictionary back to disk. Two scripts may interleave such that the first script writes to disk after the second one reads, but before it begins to write, resulting in lost work from the first script. Even worse, if the second script reads *while* the first one writes, the second script may see ill-formatted data.

To allow parallel use of the results cache, I made a new implementation backed by MongoDB [52], a document-based database that is designed to be consistent under concurrent loads. The MongoDB database is used as a simple key-value store: there is one table per model, with entries that match the format of the previous JSON file. When the multimodal recognizer needs to neuraltalk2 to score a batch of images and captions, it first queries the database for each image-caption pair, using the PyMongo bindings. If any of the pairs already have a score, it simply uses the cached score. The recognizer then scores all of the uncached pairs using neruatalk2, and writes the results back in the cache.

Bibliography

[1] Yedid Hoshen, Ron J Weiss, and Kevin W Wilson. Speech acoustic modeling from raw multichannel waveforms. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4624–4628. IEEE, 2015.

[2] S. Imai. Cepstral analysis synthesis on the mel frequency scale. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '83.*, volume 8, pages 93–96, Apr 1983.

[3] Mark Gales and Steve Young. The application of hidden markov models in speech recognition. *Foundations and trends in signal processing*, 1(3):195–304, 2008.

[4] Xiaoqiang Luo and Frederick Jelinek. Probabilistic classification of hmm states for large vocabulary continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*, volume 1, pages 353–356. IEEE, 1999.

[5] Steve J Young, Julian J Odell, and Philip C Woodland. Tree-based state tying for high accuracy acoustic modelling. In *Proceedings of the workshop on Human Language Technology*, pages 307–312. Association for Computational Linguistics, 1994.

[6] C.H. Lee, L.R. Rabiner, R. Pieraccini, and J.G. Wilpon. Acoustic modeling for large vocabulary speech recognition. *Computer Speech & Language*, 4(2):127 – 165, 1990.

[7] Ann Lee and James Glass. Mispronunciation detection without nonnative training data. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[8] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–393, 1999.

[9] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[10] Cyril Allauzen, Michael Riley, and Johan Schalkwyk. A generalized composition algorithm for weighted finite-state transducers. In *Interspeech 2009*, pages 1203–1206, 2009.

[11] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

[12] Ruslan Salakhutdinov and Geoffrey E Hinton. Deep boltzmann machines. In *International conference on artificial intelligence and statistics*, pages 448–455, 2009.

[13] Hasim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2014.

[14] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

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

[16] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[17] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren. DARPA TIMIT acoustic phonetic continuous speech corpus CDROM, 1993.

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

[19] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems*, pages 2204–2212, 2014.

[20] Kelvin Xu, Jimmy Ba, Ryan Kiros, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.

[21] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell. *CoRR*, abs/1508.01211, 2015.

[22] Y. Tabet and M. Boughazi. Speech synthesis techniques. a survey. In *Systems, Signal Processing and their Applications (WOSSPA), 2011 7th International Workshop on*, pages 67–70, May 2011.

[23] Sin-Horng Chen, Shaw-Hwa Hwang, and Yih-Ru Wang. An rnn-based prosodic information synthesizer for mandarin text-to-speech. *Speech and Audio Processing, IEEE Transactions on*, 6(3):226–239, 1998.

[24] Heiga Zen and Hasim Sak. Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4470–4474. IEEE, 2015.

[25] Pegah Ghahremani, Bagher BabaAli, Daniel Povey, Korbinian Riedhammer, Jan Trmal, and Sanjeev Khudanpur. A pitch extraction algorithm tuned for automatic speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 2494–2498. IEEE, 2014.

[26] Junichi Yamagishi. An introduction to hmm-based speech synthesis, 2006.

[27] H. Kawahara, M. Morise, T. Takahashi, R. Nisimura, T. Irino, and H. Banno. Tandem-straight: A temporally stable power spectral representation for periodic signals and applications to interference-free spectrum, f0, and aperiodicity estimation. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 3933–3936, March 2008.

[28] Kai Yu, Heiga Zen, Francois Mairesse, and Steve J Young. Context adaptive training with factorized decision trees for hmm-based speech synthesis. In *INTERSPEECH*, pages 414–417, 2010.

[29] K. Tokuda, T. Yoshimura, T. Masuko, T. Kobayashi, and T. Kitamura. Speech parameter generation algorithms for hmm-based speech synthesis. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on*, volume 3, pages 1315–1318 vol.3, 2000.

[30] Zhizheng Wu, Cassia Valentini-Botinhao, Oliver Watts, and Simon King. Deep neural networks employing multi-task learning and stacked bottleneck features for speech synthesis. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4460–4464. IEEE, 2015.

[31] Yuchen Fan, Yao Qian, Feng-Long Xie, and Frank K Soong. Tts synthesis with bidirectional lstm based recurrent neural networks. In *Interspeech*, pages 1964–1968, 2014.

[32] Yajie Miao, Mohammad Gowayyed, and Florian Metze. EESEN: end-to-end speech recognition using deep RNN models and wfst-based decoding. *CoRR*, abs/1507.08240, 2015.

[33] J. J. Godfrey, E. C. Holliman, and J. McDaniel. Switchboard: telephone speech corpus for research and development. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 1, pages 517–520 vol.1, Mar 1992.

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

[35] Anthony Rousseau, Paul Deléglise, and Yannick Esteve. Ted-lium: an automatic speech recognition dedicated corpus. In *LREC*, pages 125–129, 2012.

[36] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society, December 2011. IEEE Catalog No.: CFP11SRW-USB.

[37] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[38] Daniel P. W. Ellis. PLP and RASTA (and MFCC, and inversion) in Matlab, 2005. online web resource.

[39] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. *CoRR*, abs/1412.0035, 2014.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[41] 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.

[42] Fei Yan and K. Mikolajczyk. Deep correlation for matching images and text. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*, pages 3441–3450, June 2015.

[43] Richard Socher, Andrej Karpathy, Quoc V Le, Christopher D Manning, and Andrew Y Ng. Grounded compositional semantics for finding and describing images with sentences. *Transactions of the Association for Computational Linguistics*, 2:207–218, 2014.

[44] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[45] Polina Kuznetsova, Vicente Ordonez, Alexander C Berg, Tamara L Berg, and Yejin Choi. Collective generation of natural image descriptions. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 359–368. Association for Computational Linguistics, 2012.

[46] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Computer Vision and Pattern Recognition*, 2015.

[47] Micah Hodosh, Peter Young, and Julia Hockenmaier. Framing image description as a ranking task: Data, models and evaluation metrics. *Journal of Artificial Intelligence Research*, pages 853–899, 2013.

[48] David Harwath and James Glass. Deep multimodal semantic embeddings for speech and images. In *Proceedings of Interspeech*, 2015.

[49] R. Haeb-Umbach and H. Ney. Linear discriminant analysis for improved large vocabulary continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 1, pages 13–16 vol.1, Mar 1992.

[50] CJ Leggetter and Philip C Woodland. Speaker adaptation of continuous density hmms using multivariate linear regression. In *ICSLP*, volume 94, pages 451–454. Citeseer, 1994.

[51] Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *Transactions of the Association for Computational Linguistics*, 2:67–78, 2014.

[52] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.