Deep Learning for Automatic Speech Recognition

Ryan Hartsfield Garrett Lewellen

May 4, 2015

Introduction

TODO: Intro sentence

Due to the complexity of the speech stream and massive amount of variations possible, the problem of automatic speech recognition is a difficult one to address. The human speech system is comprised of a series of continually moving articulators with highly variable targets across contexts. Systems need to deal with all of these contexts which can cause different acoustic representations for the same lexical target. One also needs to properly handle both physical and linguistic variation across speakers, differing degrees of fluency, large vocabulary sizes, and even background noise. All of these together demonstrate the need for a robust system that can account for such variance in its acoustic model.

Deep learning approaches to ASR have seen recent success in addressing these issues, and appear to be the future means of constructing acoustic models. In this paper we examine two deep learning methods for automatic speech recognition: deep neural networks (DNNs) and convolutional neural networks (CNNs) based on the works of [DYDA12] and [AMJ+14] respectively.

We will begin by giving a brief overview of the traditional approach toward ASR using Gaussian mixture models (GMMs). In the subsequent sections, we will look in depth at the DNN approach of [DYDA12] and the CNN approach of [AMJ+14]. We will look at the overall architecture of the two models, how unsupervised pre-training is used to improve performance, and how supervised training is used to tune both systems. We will then look at experimental results of both models compared to GMMs, and follow up with thoughts on the future of ASR.

Traditional Approach

One thing that all three systems looked at here have in common is the use of hidden Markov models (HMMs) to form a hybrid structure with the acoustic model. This uses the posterior probability supplied by the acoustic model, the prior probability from a language model, and attempts to find the most probable word sequence for the acoustic input. Because these are fairly uniform across all models looked at, their inner workings and structure will be passed over here, but described in slightly more detail in the next section.

Before the rise of deep learning for automatic speech recognition, the primary tool utilized for providing the acoustic model was the Gaussian mixture model. The driving concept behind GMMs is that the acoustic spectrum can be modeled by a mixture of multivariate Gaussians. Each Gaussian is weighted and combined to form a probability distribution for the observation. These models can then be used to derive a likelihood estimate for an HMM state. Due to the popularity of this approach, mel-frequency cepstrum coefficients (MFCCs) have become the acoustic input of choice. This is because the 13 coefficients are decorrelated by the discrete cosine transform, allowing the GMM to make predictions quickly without having to train a full covariance matrix. Below is the likelihood function used to produce something similar to a posterior probability:

$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} \frac{1}{\sqrt{2\pi|\sigma_{jm}|}} \exp[(x - \mu_{jm})^T \Sigma_{jm}^{-1} (o_t - \mu_{jm})]$$

where o_t is an observation vector at time t, μ_{jm} is a mean vector for an HMM state given the multivariate m, c_{jm} is the corresponding weight, and Σ_{jm} is a covariance matrix for the same context. The likelihood function is not a true probability like what is supplied by the softmax layer in both DNNs and CNNs, but the effect is the same.

For classification, these likelihoods are transferred into the log domain for simple computation and fast evaluation. Training of GMMs is a much easier task than that of neural networks. The parameters can be estimated and re-estimated from training data using the iterative Expectation-Maximization (EM) algorithm. This allows us to continually recompute the mean, covariance, and mixture weight until we can accurately model the probability distributions of the training data.

Despite how well GMMs work, deep learning approaches have overtaken them as the acoustic model for ASR applications for a few reasons. One is simply due to an increase in processing power and

more refined algorithms for training and pre-training. But arguably the main reason why deep learning performance outstrips that of GMMs is due to the overall size of the models. For example, in TODO: cite, they use a CNN with around 60 million parameters and 650,000 neurons for image classification on the ImageNet dataset. For speech recognition, artificial neural networks are able to take in multiple frames of the acoustic context as input, while GMMs typically look at a single frame and build an expert around it. ANNs normally take in around 9-11 frames, and process the center frame around that window. This extra context allows them to take into account a variety of additional features when modeling the posterior probabilities.

Deep Neural Networks Approach

In this section we discuss the work on Large-Vocabulary Speech Recognition (LVSR) of [DYDA12] consisting of a context dependent hidden Markov model and deep neural network hybrid architecture (CD-HMM-DNN) for the acoustic model. We will begin with an overview of the general architecture, then explain the algorithms used for pre-training, and conclude with a discussion on the general procedure for training. Discussion of experimental results for this approach are deferred to the results section so that they can be compared to the convolutional neural network approach.

Architecture

TODO: Include figure of architecture?

To motivate their architecture, [DYDA12] rely on the standard noisy channel model for speech recognition presented in [JM08] where we wish to maximize the likelihood of a decoded word sequence given our input audio observations:

$$\hat{w} = \underset{w \in \mathscr{L}}{\operatorname{argmax}} \, \mathbb{P}(w|x) = \underset{w \in \mathscr{L}}{\operatorname{argmax}} \, \mathbb{P}(x|w) \, \mathbb{P}(w) \tag{1}$$

Where $\mathbb{P}(w)$ and $\mathbb{P}(x|w)$ represent the language and acoustic models respectively. [JM08] state that the language model can be computed via an N-gram approach, but [DYDA12] do not state their method, instead the authors put their efforts into explaining their acoustic model:

$$\mathbb{P}(x|w) = \sum_{q} \mathbb{P}(x, q|w) \, \mathbb{P}(q|w) \approxeq \max \pi(q_0) \prod_{t=1}^{T} a_{q_{t-1} q_t} \prod_{t=0}^{T} \mathbb{P}(x_t|q_t) \tag{2}$$

Here the acoustic model is viewed as a sequence of transitions between states of tied-state triphones which [DYDA12] refer to as senones giving us the context dependent aspect of the architecture. [**?**] explains that senones represent the pronunciation of words and are derived by decision trees. By tying triphone states together, this approach is able to avoid having to process a large number of triphones and avoid the likely sparseness of training examples for every possible triphone.

The model assumes that there is a probability $\pi(q_0)$ for the starting state, probabilities $a_{q_{t-1}q_t}$ of transitioning to the state observed at step $t-1$ to step t, and finally, the probability of the acoustics given the current state q_t. [DYDA12] expand this last term further into:

$$\mathbb{P}(x_t|q_t) = \frac{\mathbb{P}(q_t|x_t)\,\mathbb{P}(x_t)}{\mathbb{P}(q_t)}$$

Where $\mathbb{P}(x_t|q_t)$ models the tied triphone senone posterior given mel-frequency cepstral coefficients (MFCCs) based on 11 sampled frames of audio. While MFCCs come from signal processing, they have proven to be effective features for automatic speech recognition. Based on the power spectrum derived from sample audio frames, MFCCs represent characteristics of the audio that our ears are sensitive to as explained in [**?**]. $\mathbb{P}(q_t)$ is the prior probability of the senone, and $\mathbb{P}(x_t)$ can be ignored since it does not vary based on the decoded word sequence we are trying to find.

Based on this formalism, [DYDA12] chose to use pre-trained deep neural networks to estimate $\mathbb{P}(q_t|x_t)$ using MFCCs as DNN inputs and taking the senone posterior probabilities as DNN outputs. The transitioning between events is best modeled by hidden markov model whose notation, π, a, and q appears in Eqn. (2). Now that we have an overview of the general CD-DNN-HMM architecture, we can look at how [DYDA12] train their model.

Pre-Training

Given the DNN model we wish to fit the parameters of the model to a training set. This is usually accomplished by minimizing a likelihood function and deploying a gradient descent procedure to update the weights. One complication to this approach is that the likelihood can be computationally expensive for multilayer networks with many nodes rendering the approach unusable. As an alternative, one can attempt to optimize a computationally tractable surrogate to the likelihood. In this case the surrogate is the contrastive divergence method developed by [Hin02]. This sidestep enabled [HOT06] to develop an efficient unsupervised greedy pre-training process whose results can then be refined using a few iterations

of the traditional supervised backpropagation approach. In this portion of the paper we discuss the work of [Hin02] and explain the greedy algorithm of [HOT06] before going on to discuss the high-level training procedure of [DYDA12].

To understand the pre-training process, it is necessary to discuss Restricted Boltzmann Machines (RBM) and Deep Belief Networks (DBN). RBMs are an undirected bipartite graphical model with Gaussian distributed input nodes in a visible layer connecting to binary nodes in a hidden layer. Every possible arrangement of hidden, h, and visible, v, nodes is given an energy under the RBM model:

$$E(v,h) = -b^T v - c^T h - v^T W h \tag{3}$$

Where W is the weight of connections between nodes and vectors b and c correspond to the visible and hidden biases respectively. The resulting probability is then given by:

$$\mathbb{P}(v,h) = \frac{e^{-E(v,h)}}{Z} \tag{4}$$

Where Z is a normalization factor. Based on the assumptions of the RBM, [DYDA12] derive expressions for $\mathbb{P}(h=1|v)$ and $\mathbb{P}(v=1|h)$ given by:

$$\mathbb{P}(h=1|v) = \sigma(c + v^T W) \qquad \mathbb{P}(v=1|h) = \sigma(b + h^T W^T) \tag{5}$$

Where σ is an element wise logistic function. [DYDA12] argue that Eq. (5) allows one to repurpose the RBM parameters to initialize a neural network. Training of the RBM is done by stochastic gradient descent against the negative log likelihood since we wish to find a stable energy configuration for the model:

$$-\frac{\partial \ell(\theta)}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}} \tag{6}$$

however [DYDA12] point out that the gradient of the negative log likelihood cannot be computed exactly since the $\langle \cdot \rangle_{\text{model}}$ term takes exponential time. As a result, the contrastive divergence method is used to approximate the derivative:

$$-\frac{\partial \ell(\theta)}{\partial w_{ij}} \approx \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{1} \tag{7}$$

where $\langle \cdot \rangle_1$ is a single step Gibbs sampled expectation. These terms are expectations in which nodes i and j cooccur given the training data and model. Given this insight, regular stochastic gradient descent can be performed and the parameters of a RBM fitted to training data.

Now that we have an understanding of RBMs, we can shift our focus to DBNs. Deep Belief Networks are multilayer models with undirected connections between the top two layers and directed between other layers. To train these models, [HOT06] had the insight to treat adjacent layers of nodes as RBMs. One starts with the bottom two layers and trains them as though they were a single RBMS. Once those two layers are trained, then that top layer of he RBM is treated as the input layer of a new RBM with the layer above that layer acting is the hidden layer of the new RBMs. The sliding window over the layers continues until the full DBN is trained. After this, [HOT06] describe an "up-down" algorithm to further refine the learned weights. The learned parameters of this greedy approach can then be used as the parameters of a DNN as explained earlier.

Training

Training of the CD-DNN-HMM model consists of roughly a dozen involved steps. We won't elaborate here on the full details of each step, but will instead provide a high-level sketch of the procedure to convey its general mechanics.

The first high-level step of the procedure is to initialize the CD-DNN-HMM model. This is done by first training a decision tree to find the best tying of triphone states which are then used to train a CD-GMM-HMM system. Next, the unique tied state triphones are each assigned a unique senone identifier. This mapping will then be used to label each of the tied state triphones. (These identifiers will be used later to refine the DNN.) Finally, the trained CD-GMM-HMM is converted into a CD-DNN-HMM by retaining the triphone and senone structure and HMM parameters. This resulting DNN goes through the previously discussed pre-training procedure.

The next high-level step iteratively refines the CD-DNN-HMM. To do this, first the originally trained CD-GMM-HMM model is used to generate a raw alignment of states which is then mapped to its corresponding senone identifier. This resulting alignment is then used to refine the DBN by backpropagation. Next, the prior senone probability is estimated based on the number of frames paired with the senone and the total number of frames. These estimates are then used to refine the HMM transition proba-

bilities to maximize the features. Finally, if this newly estimated parameters do not improve accuracy against a development set, then the training procedure terminates; otherwise, the procedure repeats this high-level step.

Convolutional Neural Networks Approach

In this section we explore the work done in [AMJ+14] with a convolutional neural network-hidden Markov model hybrid (CNN-HMM) for phone recognition on the TIMIT dataset and a large-vocabulary voice search task. The architecture used by [AMJ+14] is heavily inspired by the use of CNNs for image recognition tasks. In this model, temporal variability is handled by the HMM, while convolutions occur along the frequency axis of the acoustic signal. We will begin by exploring the overall structure of the network, addressing the shape of the input and looking closely at the convolution and pooling plys. We will also look at the novel concept of limited weight sharing across the network. Pre-training is similar to that used in [DYDA12], so we will instead focus on the fine-tuning of parameters using the back-propagation algorithm. We will then transition into the results section where we will compare the performance of the CD-DNN-HMM with that of the CNN-HMM.

Architecture

TODO: Pic of input or architecture?

Over the past few years, convolutional neural networks have proven to be highly successful in image recognition tasks. Not only have they achieved the highest results on the MNIST character recognition datasetTODO: cite, but they've even proven to outstrip hand-crafted object recognition systems on the much more complex ImageNet dataset TODO: cite. This success primarily originates from the networks ability to replicate lower-resolution features across the input space. They have also previously been used for speech recognition in TODO: cite, but without great success. There they claim that while CNNs look promising for ASR, just taking the same structure as one built for image recognition will not suffice. For example, in that work they used convolutions performed along the time axis. [AMJ+14] attempts to address these concerns and instead design a CNN specifically for ASR.

Before discussing the network architecture itself, it's important to understand how the input to the model is structured. Contrary to the above DNN model, the CNN model used here does not use the

standard MFCC features as input. Because these are decorrelated by the discrete cosine transform, they may not maintain the locality that is necessary for the convolutions to be meaningful. As such, they used the log-energy directly from the mel-frequency spectral coefficients, what they refer to as MFSC features, along with their deltas and delta-deltas. Also, while the DNN model used only 11 frames as input, here they use 15 frames as their input window. [AMJ+14] proposes two different ways of structuring these 15 frames. These could be arranged into three 2-D feature maps with frequency on one axis and time on the other. This would allow for 2-D convolutions that normalize both frequency and time domains at the same time. Instead, they opt for the second solution where each frame is aligned into 1-D arrays across the 40 frequency bands. This results in 45 feature maps, each with 40 features. Because there are three separate feature maps for each frame, convolutions are not applied across the time domain like in TODO: cite, but instead relegated to the HMM part of the model. Instead, convolutions occur only along the frequency axis.

The network itself is constructed of 5 separate layers. After the input layer comes the convolution layer. This layer is separated into two different sub-layers called *plies*. The first being the convolution ply and the second the pooling ply. These will be discussed in depth in the following sections. Following the convolution layer are two fully-connected hidden layers with 1000 units each. The top layer of the network is a softmax output layer that provides the posterior probabilities to the HMM model. The convolution layer, despite being complex in design, actually decreases the overall model parameters by sharing weights across convolution plies. This makes it much smaller and faster to train than an equivalent DNN.

Convolution Ply

The primary goal of the convolution ply is to derive more abstracted features from only local areas of the input. In this paper, the acoustic input is convolved across the frequency axis with a specific filter size F. This means that each unit in the convolution ply draws from F adjacent frequency bands. This allows for localized features to be passed up the network. These units in the convolution ply can then be arranged in their own feature maps, where they share weights, but originate from a larger space in the lower layer. This allows for replication of features across certain frequency windows.

Because each feature map is only 1-D, a one-dimensional convolution operation is applied along the frequency axis of each frame. This creates 45 new feature maps with units confined to a limited frequency

range. Each of these units can be computed with the below formula:

$$Q_j = \sigma(\sum_{i=1}^{I} O_i \star w_{i,j})$$

where $\star$ is the convolution operator, O_i is the i-th input feature map, and $w_{i,j}$ is the local weight matrix. For each unit in the convolution ply, all input units within the filter size are multiplied by their weights, summed, and passed through a non-linear function. Without padding the input with dummy values, the convolution operation would produce a lower-dimensional feature map where each dimension decreases by the filter size $F - 1$. If this addition is done, there will be the same number of units in the input feature map as in that of the convolution ply.

Pooling Ply

The pooling ply acts as a next step from the convolution ply, where the resolution of the convolution feature maps is reduced. While it is not necessary to follow a convolution ply with a pooling ply, in this work the two always occur together. Compared to the convolution ply, the same number of feature maps are present, but their size is reduced by the pooling operation. This allows for the units in this ply to act as generalizations of the features in the convolution ply. Because the units from the convolution ply are localized and spatially located on the frequency axis, the pooled features will have slight invariance to small frequency shifts. The main problem with this is that once passed up to the pooling ply, a large amount of information about the spatial location of the feature is lost. Due to this, further convolution layers stacked on top tend to have quickly diminishing returns in their results. This is caused by extra pooling layers likely losing too much resolution to be effective.

The pooling function used in a CNN is normally either one of *averaging* or *maximization*. In image processing, TODO: cite for maxpool vs average found that max-pooling consistently provided better results over average pooling. In this work, a max-pooling function was used throughout the network:

$$p_{i,m} = \max_{n=1}^{G} q_{i,(m-1)xs+n}$$

where G is the pooling size and s is the shift size. If $G = s$, there is no overlap between pooling windows. Traditionally, these are kept from overlapping to simplify the structure of the network. TODO: cite ImageNet found that having slight overlaps in their pooling windows actually increased performance of their system. [AMJ+14] decided to vary these two values independently to discover whether or not this

held true for acoustic processing as well.

Limited Weight Sharing

In CNNs constructed for image recognition, weights are shared across units in the convolution ply to replicate features identified in a given area across the entire image. The use of weight sharing models the idea that the same feature can appear anywhere within an image. This is not the case with the speech signal, as observed features normally vary across frequency bands. Throwing away all shared weights would defeat the purpose of resorting to a CNN architecture though, so [AMJ+14] introduce the idea of weights only being shared across the same frequency bands. They refer to this as limited weight sharing (LWS). This allows the modeling of small frequency shifts within single bands. Thus instead of sharing weights across entire convolution plies, the experiment with limiting their weight sharing to convolution units that act as input to the same pooling ply. They experiment with both approaches, and find that LWS offers slight improvements over full weight sharing (FWS).

Training

Due to the depth of its architecture, pre-training of the CNN is also done using Restricted Boltzmann Machines. But in order to account for both convolution and pooling plies, some slight modifications are needed. This results in what the authors call a convolutional RBM. The chief difference from the RBMs used in [DYDA12] is that the activations in the convolution ply are stochastic. A multinomial distribution is placed over each pool, and so only a single unit can be active within the pooled set of units. This results in a requirement that there is no overlap between pooling units, or that limited weight sharing is used to isolate the convolution plies from one-another.

Once pre-training of the CNN is completed, weights across the system are fine-tuned using the error back-propagation algorithm. This is fairly straightforward in the fully-connected higher layers, but units in the convolution plies need extra modifications due to the weight sharing and sparse connections cause by the filters. Before describing how the algorithm is modified to handle these differences, we will review how training is done for the fully-connected units.

In this architecture, for the weight matrix $W^{(l)}$ of the l-th layer, the ith column is represented by $w_i^{(l)}$. For fully-connected units we start working our way backward from the final softmax output layer

L, whose output y_i is the estimated posterior probability of a given class:

$$y_i = \frac{\exp(o_i^{(L)})}{\sum_j \exp(o_j^{(L)})}$$

where $o_i^{(L)}$ is the activation computed as $o_i^{(L)} = o^{L-1} \cdot w_i^{(L)}$ without a non-linear function.

Labels for each training example are then used to perform supervised training where the goal is to minimize the cross-entropy function $Q(W^{(l)}) = -\sum_i d_i \log y_i$ for the target $\mathbf{d}$ and softmax output $\mathbf{y}$. By using the stochastic gradient descent algorithm, we can calculate the necessary weight matrix updates as:

$$\triangle W^{(l)} = \epsilon \cdot (o^{(l-1)})' e^{(l)}$$

where ϵ is the learning rate and the error vector $e^{(l)}$ is computed backwards from the final layer L:

$$e^{(L)} = d - y$$

$$e^{(l)} = (e^{(l+1)}(W^{(l+1)})') \cdot o^{(l)} \cdot (1 - o^{(l)})$$

In order to transfer this process to the convolution plies, a sparse weight matrix $\hat{W}$ is created and the input and convolution feature maps are vectorized as $\hat{o}$ and $\hat{q}$. The sparse matrix $\hat{W}$ is used to model the filter size of the convolution ply so that the activations of the convolution ply can be easily computed as $\hat{q} = \sigma(\hat{o}\hat{W})$. This places it in the same form as the activations for the fully-connected layers, and so back-propagation can be used to updates weights in the same manner as above. The main difference is that shared weights are collectively updated with the sum of their individual updates.

Because the pooling plies have no weights, no learning occurs here. Despite this, errors need to be passed back to the convolution plies the draw from. Because max-pooling is used in this work, this process is fairly straightforward. Errors are only propagated backwards to the largest unit in the group of those pooled. Units with lower activations are left alone, and the weights are unaltered.

Experimental Results

System Configurations

[DYDA12] report that their system relies on nationwide language model consisting of 1.5 million trigrams. For their acoustic model, then use a five hidden layer DNN with each layer containing 2,000 hidden units.

TODO: Add in CNN details (Full weight vs. limited weight sharing)

Datasets

TODO: Garrett

Results

	Switchboard (WER)	Bing Mobile (SER)	TIMIT (PER)
GMM	23.6	36.2	21.7
DNN	16.1	30.4	21.9
CNN	–	–	20.2
RNN	–	–	17.7

Table 1: title

Direct comparison of the two systems is complicated by the fact that both papers report different metrics against different datasets. [DYDA12] reports a sentence level accuracy rate, while [AMJ+14] reports the phone error rate.

Conclusions

TODO: Garrett TODO

References

[AMJ+14] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech & Language Processing*, 22(10):1533–1545, 2014.

[DYDA12] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech & Language Processing*, 20(1):30–42, 2012.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[JM08] Daniel Jurafsky and James H. Martin. *Speech and Language Processing, 2nd Edition.* Prentice Hall, 2008.

[LMM+14] Haizhou Li, Helen M. Meng, Bin Ma, Engsiong Chng, and Lei Xie, editors. *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014.* ISCA, 2014.