Comparing a High and Low-Level Deep Neural Network Implementation for Automatic Speech Recognition

Jessica Ray
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA
jessica.ray@ll.mit.edu

Brian Thompson
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA
brian.thompson@ll.mit.edu

Wade Shen
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA
swade@ll.mit.edu

ABSTRACT
The use of deep neural networks (DNNs) has improved performance in several fields including computer vision, natural language processing, and automatic speech recognition (ASR). The increased use of DNNs in recent years has been largely due to performance afforded by GPUs, as the computational cost of training large networks on a CPU is prohibitive. Many training algorithms are well-suited to the GPU; however, writing hand-optimized GPGPU code is a significant undertaking. More recently, high-level libraries have attempted to simplify GPGPU development by automatically performing tasks such as optimization and code generation. This work utilizes Theano, a high-level Python library, to implement a DNN for the purpose of phone recognition in ASR. Performance is compared against a low-level, hand-optimized C++/CUDA DNN implementation from Kaldi, a popular ASR toolkit. Results show that the DNN implementation in Theano has CPU and GPU runtimes on par with that of Kaldi, while requiring approximately 95% less lines of code.

General Terms
Machine learning, Automatic optimization, Automatic speech recognition

Keywords
Python, Theano, DNN, Kaldi, CUDA, GPU

1. INTRODUCTION
Deep neural networks (DNNs) have provided significant performance gains over the previous state-of-the-art in a number of tasks across a variety of fields such as computer vision [11], natural language processing [4], and automatic speech recognition (ASR) [9].

Neural networks have been a topic of research for many years [12], but the application of large networks (such as DNNs) was limited by the huge computational cost of training such networks on a CPU. Recently, the use of DNNs has increased substantially in part due to the computational power afforded by GPUs. Many training algorithms, such as backpropagation [17], are well suited for GPUs because they can be represented as highly parallelizable matrix operations. However, even with the development of NVIDIA CUDA [14], developing for the GPU equates to writing and hand-optimizing a significant amount of code. Furthermore, in many cases a CPU implementation must be developed in parallel with the GPGPU implementation and maintained simultaneously.

Recently, high-level libraries have attempted to simplify the GPGPU coding process by automatically performing tasks such as optimizations and code generation, allowing the programmer to focus on algorithmic implementation instead of platform-specific development. We consider one such library, Theano [2], in the context of implementing a DNN for ASR. Theano is a Python library that optimizes symbolic expressions, performs symbolic differentiation, and generates both CPU and GPU code automatically. The specific ASR task considered in this work is phone recognition on the TIMIT speech corpus [6].

We compare our Theano DNN implementation to a hand-optimized C++/CUDA DNN implementation [18] from the popular Kaldi ASR toolkit [15]. We find that our Theano implementation is able to match the runtimes of the Kaldi implementation on both the CPU and GPU, indicating that the automatic optimizations performed by Theano are on par with those done by hand in Kaldi. Furthermore, the codebase for the Theano implementation has approximately 95% less lines of code then the C++/CUDA implementation in Kaldi, suggesting lower development and maintenance times.

The remainder of this paper is organized as follows: we begin with a brief introduction to ASR (Section 2), followed by an introduction to DNNs and their applications to ASR (Section 3). In Section 4, we contrast the Theano and Kaldi implementations. In Sections 5 and 6, we outline the TIMIT phone recognition experiments performed and present ASR accuracy and runtime performance results.
2. ASR OVERVIEW

The goal of ASR is to provide automatic transcriptions for speech. This is achieved using statistical models known as the acoustic model and the language model; this work considers only the acoustic model. The acoustic model describes the individual units of speech (phones) extracted from the raw speech signal. The model is trained in a supervised fashion using speech and associated transcripts. From a mathematical standpoint, given a set of observations, \( o \), we want to find the most likely sequence of words, \( w \), for those observations. Using Bayes’ Rule, this can be written as

\[
p(w|o) \propto p(o|w)p(w)
\]

where \( p(o|w) \) is estimated by the acoustic model and \( p(w) \) is estimated by the language model.

2.1 Acoustic Model Training

In a traditional ASR system, the raw speech signal is first processed and converted into a series of feature vectors to be used in training. These feature vectors are extracted from overlapping time slices (speech frames) in the speech signal. Once all the feature vectors have been extracted, they are used to train a series of hidden Markov models (HMMs) [16] via the Baum-Welch algorithm [1]. One HMM is trained per phone and individual HMMs are connected to form larger units of speech. The emission probabilities of the HMM states are modeled via Gaussian mixture models (GMMs).

2.2 Decoding

Once the acoustic and language models are trained, a decoding process can be applied to unknown speech to generate transcripts. Formally, we want to find the most likely sequence of words \( w^* \) for a set of observations \( o \):

\[
w^* = \arg \max_w p(o|w)p(w)
\]

Decoding is not limited to words; phone decoding, as discussed later in this paper, can be performed in the same manner.

3. DNN OVERVIEW

An feed-forward artificial neural network is an architecture comprised of multiple layers of non-linear transformations, where each layer is made up of a series of nodes. These architectures consist of an input layer, an output layer (often representing posterior probabilities), and some number of hidden layers in between. Feed-forward refers to the fact that information can only flow in one direction through the network. A DNN is simply a feed-forward neural network with one or more hidden layers [9]. Figure 1 shows an example DNN with four layers. Each hidden layer of a DNN represents a different abstraction of the input data, with the higher layers (closer to the output layer) having more abstract representations than the lower layers. In typical training, rather than specify what each layer should represent, the learning algorithm is free to try to discover the optimal abstractions for a given task.

3.1 Forward Propagation

A procedure known as forward propagation is used to calculate DNN output values for a given input vector. First, the input layer is set equal to the input vector. Next, using the weight connections between the input and first hidden layers and a bias value for each hidden node, the values of the nodes in the first layer are calculated. This procedure repeats for each successive layer, using the node values computed in the previous layer. The values for each layer are calculated as follows:

\[
x^{i+1} = \sigma(W^i x^i + b^{i+1})
\]

where \( x^i \) is the vector of values for the \( i^{th} \) layer, \( W^i \) is the matrix of weight connections between layer \( i \) and \( i+1 \), \( b^i \) is the vector of biases for the \( i^{th} \) layer, and \( \sigma() \) is an elementwise non-linear transform, typically a sigmoid. \( i \) is the layer index and ranges from 0 to \( N-1 \), where \( N \) is the total number of layers in the network. If the output layer is used for estimating posterior probabilities, the \( \text{softmax}() \) activation function is applied in place of \( \sigma() \) on the output layer:

\[
\text{softmax}(x) \equiv \frac{exp(x)}{\sum_k exp(x[k])}
\]

3.2 Backpropagation

A DNN must be trained before it can be used to calculate output values. Training a DNN consists of finding weights and biases that minimize a loss function. An algorithm commonly used for training is backpropagation, which iteratively updates the weights and biases. When training a network to produce posterior probabilities, the standard loss function to use is cross entropy. A thorough explanation of backpropagation is given in [17], and a summary of the procedure is given here:

1. Initialize weights and biases to small random values \(^1\)
2. Propagate a training vector through the network using forward propagation
3. Compute the errors at the output layer
4. Compute the gradients of the errors with respect to each weight and bias using the chain rule to propagate errors backwards through the network
5. Update weights and biases using the computed gradients, scaled by a learning rate to control the size of the update
6. Repeat steps 2-5 until some stopping criterion is met

\(^1\)Other initialization methods exist [5, 13]
cross-validation error, which is the average error of the network on a held-out set.

The update method described above is stochastic gradient descent, meaning that the gradients are computed and each weight update is performed once per training vector. This equates to a series of matrix-vector operations for each iteration. In practice, minibatch stochastic gradient descent (MSGD) is typically used. With MSGD, the gradient is computed over a set of training vectors (called the minibatch). One weight update is performed per minibatch via highly parallelizable matrix-matrix operations, which are well-suited for GPU processing.

### 3.3 Applying DNNs to ASR

We use DNNs to estimate the posterior probabilities of phone-states (a further subdivision of a phone), and then use these estimates to perform phone decoding. This type of configuration, in which the DNN replaces the GMM, is known as a hybrid system [3]. Another common configuration is a tandem system [8], where DNN outputs are used as features for training a new GMM-HMM system.

Human transcriptions typically consist of words, as opposed to the targets (such as phone-states) needed to train a DNN. Furthermore, training requires a target for each frame (once every 10ms in our system), while transcriptions typically only delineate word or segment boundaries. For these reasons, an alignment is performed prior to training a DNN. During alignment, words are replaced with sequences of phones, which are further divided into phone-states. The phone-states are then time aligned with the audio frames. An external dictionary is used to provide possible word-to-phone mappings, and an existing GMM-HMM system is used for selecting the best phone sequence and determining the time boundaries of the phone-states.

### 4. DNN IMPLEMENTATIONS

We implemented a DNN in Python using the Theano library and integrated it into our existing HTK-based [19] ASR system. Theano allows a user to define symbolic expressions using syntax similar to NumPy. The symbolic expressions are statically typed, allowing Theano to perform automatic symbolic optimizations and differentiation. Optimized expressions are compiled to machine code for both the CPU and GPU.

Our DNN implementation contains ~800 lines of code. We contrast this with the original Kaldi C++/CUDA DNN implementation, written by Karel Vesely [18]. This system has ~5,200 lines of code, plus an additional ~12,000 lines for Kaldi’s CUDA Matrix library, which is shared with Dan Povey’s DNN implementation [20]. Karel’s implementation includes pretraining and several discriminative training methods which are not included in our Theano implementation. Files clearly dedicated to these features were excluded from the code counts.

Our implementation contains significantly less lines of code than Karel’s because Theano handles many necessary details automatically. This difference is clearly seen in the weight and bias update step of backpropagation. Compared to the native implementation (Figure 2), the updates are very similar in Theano (Figure 3) due to its automatic symbolic differentiation (T.grad()) and transparent use of the GPU. Note, comments and white space have been removed from the code fragments for brevity.

### Code fragments computing the weight and bias updates with Kaldi

```cpp
/// Fragment from function ‘Backpropagation'
int32 i = components_.size()-1;
components_->back()->
Backpropagate(propagate_buf_[i],
propagate_buf_[i+1], out_diff,
&backpropagate_buf_[i-1]);
if (components_[i]->IsUpdatable()) {
    UpdatableComponent *uc =
        dynamic_cast<UpdatableComponent*>(
            components_[i]);
    uc->Update(propagate_buf_[i], out_diff);
}

/// Gradient/weight update function
void Update(const CuMatrix<BaseFloat> &input,
    const CuMatrix<BaseFloat> &diff) {
    const BaseFloat lr = opts_.learn_rate;
    scale_data_grad_.Resize(InputDim(), kSetZero);
    CuMatrix<BaseFloat> gradient_aux(diff);
    gradient_aux.MulElements(input);
    scale_data_grad_.AddRowSumMat(1.0,
        gradient_aux, 0.0);
    scale_data_grad_.AddVec(-lr, scale_data_grad_);
}
```

Figure 2: Code fragments computing the weight and bias updates with Kaldi.
import theano.tensor as T

#### In class HiddenLayer
def backp_loss(self, y):
    return -T.mean(T.log(self.p_y_given_x)
                     - T.log(y.shape[0]), y)

#### In class DNN
self.backp_error = self.output.backp_loss(self.y)
gradients = T.grad(self.backp_error, self.weights)
for weight, gweight in zip(self.weights, gradients):
    update = weight - gweight * learning_rate

Figure 3: Code fragments computing the weight and bias updates with Theano

5. EXPERIMENTAL SETUP

To compare training speeds with Theano and Kaldi on the CPU and GPU, we use the TIMIT speech corpus [6]. We use the 462 speaker training set and the 24 speaker core test set. A validation set of 50 speakers was randomly selected from the remaining test data not included in the core test set. The small size of this corpus allows completion of all runs on the CPU, as a more realistic size data set would require weeks or months to train each network on the CPU. Human-annotated frame boundaries were provided with TIMIT, however, we used an existing GMM-HMM system to generate the alignments.

DNNs were trained using 52-component feature vectors consisting of 12 Perceptual Linear Prediction coefficients [7], along with the zeroth coefficient and first, second, and third order differentials. Feature extraction was performed using HTK. Feature vectors from 13 successive frames were combined into a single DNN input vector of length 676. The output layer of the DNN consisted of 115 phone-state targets. We used a minibatch size of 128 and an initial learning rate of 0.1, unless otherwise stated. We implemented the “newbob” learning rate schedule and stopping criterion [10].

For our Theano implementation, we used Theano v0.6rc3, Python v2.7.6, and NumPy v1.8.0. CPU experiments were performed on machines equipped with four Intel Xeon E7-L8867 processors and 512GB RAM. Both Theano and Kaldi were configured to use openBLAS 0.2.8 running with 20 threads. For the GPU experiments, we used machines equipped with NVIDIA Tesla K40m cards running CUDA 5.5. All GPU operations were single-precision and runs were performed on a single card. Due to the small size of TIMIT, all training, validation, and testing data was loaded into memory upon startup for both CPU and GPU experiments.

6. RESULTS

Figure 4 illustrates the advantage of using DNNs with many hidden layers for phone decoding on the TIMIT corpus. Phone error rate, as computed with NIST’s scilce scoring software \(^2\), is shown for networks containing one to eight hidden layers. Hidden layers contained 1024, 2048, or 3072 nodes, with layer size held constant for a given network. The number of hidden layers and layer sizes were selected to coincide with those in [13]. The best performing network was the largest, with eight hidden layers of size 3072. This gave a PER of 23.8. Compared to our GMM-HMM system, which gets a PER of 29.3, this is a 18.8% relative reduction.

6.1 Average Backpropagation and Cross-Validation Time

The average CPU and GPU time per iteration of backpropagation (Figure 5) and cross-validation (Figure 6) are shown for both Theano and Kaldi. The networks have one to eight hidden layers, with 3072 nodes per hidden layer. Experiments were also run using 1024 and 2048 nodes per hidden layer, but results are not shown as they were consistent with those in Figures 5 and 6.

For both backpropagation and cross-validation, the GPGPU implementation is much faster than the CPU implementation, as expected. Theano and Kaldi have nearly identical runtimes on the CPU and Theano slightly outperforms Kaldi on the CPU. This suggests that Theano’s automatic optimizations can replace the hand-optimizations implemented in the native C++ and CUDA code. Note that the number of iterations of backpropagation and cross-validation per experiment typically ranged from 25 to 30.

After running our initial experiments, we swept through a series of minibatch sizes and initial learning rates with Theano to see the effect on the PER. The experiments were run on the GPU, enabling us to perform all 28 experiments in a matter of hours. We can see in Figure 7 that a minibatch size of 64 and initial learning rate of 0.5 produced better results than our original configuration.

7. CONCLUSION

We demonstrated that a DNN implemented in Theano approximately matched the runtime performance of a hand-optimized, low-level implementation for both the CPU and GPU on the TIMIT phone recognition task. Furthermore, the use of Theano decreased lines of code from ~17,000 down to ~800. This suggests both a significant decrease in development time and a resulting codebase which is easier to maintain and modify. Overall, the success of Theano in this work implies that hand-optimizing low-level code may be unnecessary for many applications, even when targeting GPUs.

\(^2\)http://www.itl.nist.gov/iad/mig/tools/
8. ACKNOWLEDGEMENTS

The authors would like Elizabeth Godoy, Doug Reynolds, and Fred Richardson for their helpful suggestions and feedback.

9. REFERENCES


