An Efficient Phone N-gram Forward-backward Computation Using Dense Matrix Multiplication

Khe Chai Sim, Arun Narayanan
Google Inc., USA
khechai@google.com, arunnt@google.com

Abstract
The forward-backward algorithm is commonly used to train neural network acoustic models when optimizing a sequence objective like MMI and mMBR. Recent work on lattice-free MMI training of neural network acoustic models shows that the forward-backward algorithm can be computed efficiently in the probability domain as a series of sparse matrix multiplications using GPUs. In this paper, we present a more efficient way of computing forward-backward using a dense matrix multiplication approach. We do this by exploiting the block-diagonal structure of the n-gram state transition matrix; instead of multiplying large sparse matrices, the proposed method involves a series of smaller dense matrix multiplications, which can be computed in parallel. Efficient implementation can be easily achieved by leveraging on the optimized matrix multiplication routines provided by standard libraries, such as NumPy and TensorFlow. Runtime benchmarks show that the dense multiplication method is consistently faster than the sparse multiplication method (on both CPUs and GPUs), when applied to a 4-gram phone language model. This is still the case even when the sparse multiplication method uses a more compact finite state model representation by excluding unseen n-grams.

Index Terms: forward-backward algorithm, sequence training

1. Introduction
Forward-backward is an important algorithm used in many speech and natural language processing tasks, including training of hidden Markov models (HMMs) [1], and computing the derivatives for sequence losses, such as Maximum Mutual Information (MMI) [2]. Minimum Phone Error (MPE) [3] and Minimum Bayes Risk (MBR) [4, 5, 6]. For sequence training, the forward-backward algorithm is typically applied to compute the necessary statistics from lattices [2, 3, 7].

Recently, lattice-free MMI training of neural network acoustic models, where the denominator statistics needed to compute the MMI loss are collected from a phone n-gram language model, has been successfully applied to large vocabulary speech recognition systems [8, 9]. The denominator statistics can be computed efficiently using the forward-backward algorithm in the probability domain, as a series of sparse matrix multiplications. This makes it easy to take advantage of existing optimized matrix multiplication routines, including GPU support, to improve training time.

In this paper, we propose a dense matrix multiplication method for computing the forward-backward probabilities, that takes advantage of the block-diagonal structure of the state transition matrix of an n-gram model. By using a more compact representation of the state transition matrix in the form of a dense multi-dimensional array, each forward and backward step can be computed more efficiently as parallel multiplication of smaller matrices. This forward-backward computation method can be easily implemented using standard numerical computation software, such as NumPy [10] and TensorFlow [11].

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the forward-backward algorithm. Sections 3 and 4 present the sparse and dense matrix multiplication methods, respectively. Section 5 describes how the dense matrix multiplication method can be implemented efficiently using NumPy or TensorFlow. Section 6 presents runtime benchmark results using 4-gram phone language models.

2. Forward Backward Algorithm
The forward-backward algorithm can be used to efficiently compute, \( P_b(X_t^T, q_t = j) \), the joint probability of a finite-state model, \( \theta \), generating the data, \( X_t^T = \{x_1, x_2, \ldots, x_T\} \), and being in state \( j \) at time \( t \). This joint probability is useful for computing the likelihood of the model generating the data:

\[
P_b \left( X_t^T \right) = \sum_{j=1}^{S} P_b \left( X_t^T, q_t = j \right)
\]

as well as the state-alignment probabilities:

\[
P_b \left( q_t = j | X_t^T \right) = \frac{P_b \left( X_t^T, q_t = j \right)}{\sum_{i=1}^{S} P_b \left( X_t^T, q_t = i \right)}
\]

where \( S \) is the total number of states. Efficient computation is achieved by decomposing the joint probability into the forward and backward probabilities:

\[
P_b \left( X_t^T, q_t = j \right) = P_b \left( X_1^T, q_t = j \right) P_b \left( X_{t+1}^T | q_t = j \right)
\]

where the forward and backward probabilities can be computed efficiently using the following recursions:

\[
P_b \left( X_1^T, q_t = j \right) = \alpha_t[j] = \sum_{i=1}^{S} b_t[i] A[i, j] \alpha_{t-1}[i]
\]

\[
P_b \left( X_{i+1}^T | q_t = j \right) = \beta_{i+1}[j] = \sum_{i=1}^{S} b_{i+1}[i] A[i, j] \beta_{i+1}[i]
\]

where \( \alpha_t[j] \), \( \beta_{i+1}[j] \), \( A[i, j] \) and \( b_t[i] \) are the forward probability, backward probability, state transition probability and observation probability, respectively:

\[
\alpha_t[j] = P_b \left( X_1^T, q_t = j \right)
\]

\[
\beta_{i+1}[j] = P_b \left( X_{i+1}^T | q_t = j \right)
\]

\[
A[i, j] = P_b \left( q_t = i | q_{t-1} = j \right)
\]

\[
b_t[i] = P_b \left( x_t | q_t = i \right)
\]
Traditionally, the recursions are computed in the logarithmic domain to avoid numerical underflow. Recent work [8] has shown that, with some care, the forward-backward recursions can be computed directly in the probability domain, as a series of sparse matrix multiplications.

3. Sparse Matrix Multiplication

It is easy to show that the forward and backward steps can be computed as a series of sparse matrix multiplications by expressing Eqn. 3 and 4 in vector form as follows:

\[ \alpha_t = b_t \odot (A \times \alpha_{t-1}) \]  \hspace{1cm} (9)

\[ \beta_t = A^\top \times (b_{t+1} \odot \beta_{t+1}) \]  \hspace{1cm} (10)

where \( \odot \) denotes an element-wise multiplication operator and \( \times \) is a matrix multiplication operator. The various probabilities in vector form are given by:

\[ \alpha_t = [ \alpha_t[1], \alpha_t[2], \ldots, \alpha_t[S] ]^\top \]  \hspace{1cm} (11)

\[ \beta_t = [ \beta_t[1], \beta_t[2], \ldots, \beta_t[S] ]^\top \]  \hspace{1cm} (12)

\[ b_t = [ b_t[1], b_t[2], \ldots, b_t[S] ]^\top \]  \hspace{1cm} (13)


For an n-gram model with V symbols and order \( n > 0 \), there are \( V^n \) permissible transitions. Therefore, \( A \) is a sparse \( S \times S \) matrix with only \( V^n \) nonzero entries, corresponding to the n-gram probabilities. The density of \( A \) is given by

\[ \frac{V^n}{S^2} = \frac{V^n}{(2n-1)} = V^{2-n} \]  \hspace{1cm} (15)

Note that for a bigram model, \( A \) is a dense matrix. As \( n \) increases, the density of \( A \) decreases exponentially.

4. Dense Matrix Multiplication

The transition matrix, \( A \), of an n-gram model has the following properties that can be exploited to design an efficient implementation of the forward-backward algorithm:

- There are \( V^n \) nonzero entries, corresponding to the n-gram probabilities;
- There are \( V \) nonzero entries in each row, corresponding to the outgoing transitions from each state;
- There are \( V \) nonzero entries in each column, corresponding to the incoming transitions from each state;

Given the above properties, the rows (or columns) of \( A \) can be rearranged to form a block-diagonal matrix, such that there are \( V^{n-2} \) blocks and each block is a \( V \times V \) dense matrix. To better understand the structure of \( A \), it is convenient to represent the state transition as an \( n \)-dimensional tuple:

\[ \bar{q}_{i-1} = (u_0, u_2, \ldots, u_{n-1}, u_n) \]  \hspace{1cm} (16)

Here, the source and target states are \((n-1)\)-dimensional tuples (explicitly representing the symbol sequence history), instead of a scalar index:

\[ \bar{q}_{i-1} = u_{2:n} \hspace{1cm} (6) \]

\[ \bar{q}_i = u_{1:n-1} \hspace{1cm} (7) \]

Note that the first \( n-2 \) elements of \( \bar{q}_{i-1} \) are the same as the last \( n-2 \) elements of \( \bar{q}_i \). This representation is more compact as it conveniently eliminates invalid transitions, keeping only those that satisfy the constraints \( \bar{q}_{i-1}[j] = \bar{q}_i[j+1] \) for \( 1 \leq i < n-2 \). As such, Eqn. 3 and 4 can be written with tuple indexing as follows:

\[ \alpha_t[u, \ldots] = \sum_{u=1}^{V} b_t[u, \ldots] A[u, \ldots] \alpha_{t-1}[u, \ldots] \]  \hspace{1cm} (18)

\[ \beta_t[u, \ldots] = \sum_{u=1}^{V} b_{t+1}[u, \ldots] A[u, \ldots] \beta_{t+1}[u, \ldots] \]  \hspace{1cm} (19)

Note that the sums are defined over \( u \in [1, V] \), instead of \( i \in [1, S] \). These equations are defined for all possible values of \( u_{1:n-2} \) (there are \( V^{n-2} \) of them), as denoted by the shorthand, ‘..’, in the above equations. Note the similarities between Eqn. 18, 19 and Eqn. 3, 4. Therefore, Eqn. 18, 19 can be written in matrix form, similar to Eqn. 9, 10, as

\[ \bar{\alpha}_t = b_t \odot (\bar{A} \times \bar{\alpha}_{t-1}) \]  \hspace{1cm} (20)

\[ \bar{\beta}_t = \bar{A}^\top \times (b_{t+1} \odot \bar{\beta}_{t+1}) \]  \hspace{1cm} (21)

where \( \bar{\alpha}_t, \bar{\beta}_t \) and \( \bar{b}_t \) are \( V \)-dimensional vectors, whose \( v \)-th elements are given by \( \alpha_t[v, \ldots], \beta_t[v, \ldots] \) and \( b_t[v, \ldots] \), respectively. \( \bar{A} \) is a \( V \times V \) matrix whose \((u, v)\) element is given by \( A[u, \ldots, v] \). As a result, each forward and backward step can be computed as \( V^{n-2} \) independent matrix multiplications of size \( V \times V \).

The state transition probabilities are now represented by an \( n \)-dimensional array, \( \bar{A} \), whose elements are indexed by the state transition tuple:

\[ \bar{A}[\bar{u}] = \bar{P}_0 (\bar{q}_i = \bar{u}_{1:n-1} | \bar{q}_{i-1} = \bar{u}_{2:n} ) \]  \hspace{1cm} (22)

\( \bar{A} \) is a dense array that conveniently holds the \( V \times V \) block matrices, \( \bar{A} \), in its first and last dimensions \((\bar{A}[i], \ldots]) \). The corresponding forward, backward and state observation probabilities are represented as \((n-1)\)-dimensional arrays, \( \bar{\alpha}_t, \bar{\beta}_t \), and \( \bar{b}_t \) respectively. 1 The elements of these arrays can be indexed by the state tuple, \( \bar{q}_i \), as follows:

\[ \alpha_t[\bar{q}] = \bar{P}_0 (X_t^\top, \bar{q}_i = \bar{q}) \]  \hspace{1cm} (23)

\[ \beta_t[\bar{q}] = \bar{P}_0 (X_{t+1}^\top, \bar{q}_i = \bar{q}) \]  \hspace{1cm} (24)

\[ b_t[\bar{q}] = \bar{P}_0 (x_i(\bar{q}_i = \bar{q}) \]  \hspace{1cm} (25)

The corresponding array expressions for Eqn. 18 and 19 are given by:

\[ \bar{\alpha}_t = b_t \odot (\bar{A} \odot \bar{\alpha}_{t-1}) \]  \hspace{1cm} (26)

\[ \bar{\beta}_t = \bar{A} \odot (b_{t+1} \odot \bar{\beta}_{t+1}) \]  \hspace{1cm} (27)

where \( \odot \) and \( \odot \) denote the parallel matrix multiplications for the forward and backward steps, respectively, according to Eqn. 18 and 19. These can be easily computed using standard numerical computation packages as described in the next section. Since the state transition matrix, \( \bar{A} \), is a dense array, this approach is referred to as dense matrix multiplication.

\[ \footnote{1}{In practice we use an acoustic state to n-gram state map to obtain observation probabilities.} \]

1647
4.1. Handling Self Loops

In automatic speech recognition, self loops are typically added to an n-gram model to accommodate variable symbol durations, by allowing each symbol to consume more than one speech frames. Self loops can be easily handled by the sparse matrix multiplication by adding the self-loop probabilities to the leading diagonal elements of the sparse state transition matrix, $A$. However, it is not possible to incorporate self loops into the dense version of the state transition matrix, as it violates the conditions described in Section 4. It is better to consider the self loops separately, which involves modifying Eqn. 26 and 27 as follows:

$$\alpha_t = b_t \odot [(1 - \rho)A \odot \alpha_{t-1} + \rho \alpha_{t-1}]$$  
(28)

$$\beta_t = (1 - \rho)A^\top \odot \beta_{t+1} + \rho \beta_{t+1}$$  
(29)

where $\rho$ is the self-loop probability and $\hat{\beta}_{t+1} = b_{t+1} \odot \beta_{t+1}$.

5. Implementations

5.1. Dense Matrix Multiplication

Both NumPy and TensorFlow support efficient matrix multiplication of multi-dimensional arrays with broadcasting [10], that is particularly suited for the computation of the forward and backward steps. The `matmul` method in both packages performs a 2-dimensional matrix-matrix multiplication on the last two dimensions (axes). Therefore, by carefully transposing the arrays with a certain permutation, the forward and backward algorithms can be easily computed. For the forward and backward probabilities to have compatible shapes, they are also represented using $n$-dimensional arrays where the size of the last dimension is $1^2$.

The necessary transpositions required to compute the forward step in Eqn. 18 are indicated by the arrows in the figure below (excluding observation probabilities for brevity):

$$A[j, \ldots, i]$$

$\alpha_t[i, \ldots, j, 1] = A[i, \ldots, j, i] \times \alpha_{t-1}[i, \ldots, j, 1]$

$\alpha_t[j, \ldots, i]$

The first dimension of the transition probability array is moved to the next-to-last dimension. This can be precomputed, as they are independent of the inputs. The result from the multiplication needs to be transposed by moving the next-to-last dimension back to the first dimension. Similarly, for the backward step, the following transpositions are needed:

$$A[j, \ldots, i]$$

$\beta_{t+1}[j, \ldots, i] = A[j, \ldots, i, j] \times \beta_{t+1}[j, \ldots, i, 1]$

$\beta_{t+1}[j, \ldots, i]$

The state transition array is transposed such that the first dimension is moved to the last dimension. The backward probability array from the previous step needs to be transposed such that the first dimension is moved to the next-to-last dimension, before the multiplication.

5.2. Static vs. Dynamic Loop

In TensorFlow, the forward and backward recursions can be computed in two ways. One way is to construct the TensorFlow computation graph by explicitly unfolding the recursion operations. This requires the length of the sequences to be determined when creating the graph; the resulting graph only works for fixed length sequences. To cope with sequences with different lengths, it is necessary to create a graph that works for a maximum supported length and padding the shorter sequences accordingly. Alternatively, TensorFlow also supports a dynamic way of unfolding the recursion when the actual computation is performed. This allows a single graph to be used for sequences of arbitrary length. As shown later in Section 6, using a dynamic loop also leads to better runtime performance.

6. Experiments

To compare the effectiveness of the various implementations of the forward-backward algorithm, we first ran some simulation experiments using a randomly generated n-gram language model of order 4 and with 42 symbols, which are typical numbers for English. The n-gram probabilities are obtained based on the n-gram counts, randomly sampled from a uniform distribution between 0 and 1. There are a total of 3,111,696 (42$^4$) n-gram probabilities. Each forward/backward probability vector has 74,088 (42$^3$) elements. The observation probabilities are also randomly generated from a uniform distribution between 0 and 1. For the simulation, we perform the forward-backward computation over 100 observation probability sequences, each of length $T = 200$. The runtime benchmark is based on the average time taken (in seconds) to perform forward-backward on each 200-frame long sequence, on a machine with 12 Intel Xeon CPU processors at 3.5 GHz and a Quadro K620 GPU. In this paper, we considered six different implementations:

- NpSparse: a NumPy implementation of the sparse matrix multiplication approach, as described in Section 3.
- NpDense: a NumPy implementation of the dense matrix multiplication approach, as described in Section 4.
- TfStaticSparse: a TensorFlow implementation of the sparse matrix multiplication approach using a static loop, as described in Section 5.1.
- TfDynamicSparse: a TensorFlow implementation of the sparse matrix multiplication approach using a dynamic loop, as described in Section 5.1.
- TfStaticDense: a TensorFlow implementation of the dense matrix multiplication approach using a static loop.
- TfDynamicDense: a TensorFlow implementation of the dense matrix multiplication approach using a dynamic loop.

Fig. 1 compares the runtime performance of the above implementations, without and with using GPU hardware acceleration, for the TensorFlow implementations. The figure clearly shows that the dense matrix multiplication approach is consistently faster than the sparse matrix multiplication counterpart. For the NumPy implementations, the speed-up factor for the dense version is 1.47. The corresponding speed-up factors for the TensorFlow implementations using dynamic loop are 5.23 and 0.55, when running on CPU and GPU, respectively. The TensorFlow implementation of the sparse matrix multi-
In this paper, we compared the runtime performance of the sparse and dense matrix multiplication methods for computing the n-gram forward-backward probabilities. The latter exploits the block-diagonal structure of the permuted state transition matrix, thereby lending the forward and backward steps to be computed more efficiently as independent multiplications than for the dense method. We show that efficient implementation of both the methods can be achieved by taking advantage of the optimized matrix multiplication routines and GPU support provided by the TensorFlow computation software. Our runtime benchmark experiments using simulated and actual 4-gram phone language models reveal that the proposed dense method consistently outperforms the sparse method, both when using CPU and GPU for computation. Our fastest implementation, using the dense matrix multiplication method using CPU, achieved a forward-backward computation time of 0.85 seconds for a 200-frame long observation sequence.

7. Conclusions

and TfDynamicDense (CPU and GPU) methods using randomly generated n-gram language models with different densities. As expected, the runtime of the TfDynamicDense method stays constant regardless of the density, while that of the TfDynamicSparse method increases linearly with density. The dense method, when run on GPU, is as fast as the sparse method when the density of the state transition matrix is about 53%. The break-even point for running the dense method on CPU is at about 32% density.

Finally, we compared the runtime of TfDynamicDense and TfDynamicSparse using a phone 4-gram backoff language model trained on phone transcription of 22 million anonymous, hand-transcribed voice-search utterances. The phone transcriptions are obtained by force-aligning the word-level transcriptions with the audio. The resulting model has 42 unigrams, 1,634 bigrams, 42,372 trigrams and 639,825 4-grams. The full 4-gram model has 74,088 states and 3,039,372 state transitions. A more compact representation of the model that keeps only the seen n-grams has 44,088 states and 1,803,673 state transitions. Fig. 3 compares the runtimes of the TfDynamicDense method using the full model and the TfDynamicSparse method using both the full and compact models. The TfDynamicDense method running on CPU achieved the best runtime performance of 0.85 seconds per sequence, 1.77 times faster than the TfDynamicSparse method running on GPU with the compact model.
8. References


