IMPROVING THE EFFICIENCY OF FORWARD-BACKWARD ALGORITHM USING BATCHED COMPUTATION IN TENSORFLOW

Khe Chai Sim, Arun Narayanan, Tom Bagby, Tara N. Sainath, Michiel Bacchiani

Google Inc., USA

ABSTRACT

Sequence-level losses are commonly used to train deep neural network acoustic models for automatic speech recognition. The forward-backward algorithm is used to efficiently compute the gradients of the sequence loss with respect to the model parameters. Gradient-based optimization is used to minimize these losses. Recent work has shown that the forward-backward algorithm can be efficiently implemented as a series of matrix operations. This paper further improves the forward-backward algorithm via batched computation, a technique commonly used to improve training speed by exploiting the parallel computation of matrix multiplication. Specifically, we show how batched computation of the forward-backward algorithm can be efficiently implemented using TensorFlow to handle variable-length sequences within a mini batch. Furthermore, we also show how the batched forward-backward computation can be used to compute the gradients of the connectionist temporal classification (CTC) and maximum mutual information (MMI) losses with respect to the logits. We show, via empirical benchmarks, that the batched forward-backward computation can speed up the CTC loss and gradient computation by about 183 times when run on GPU with a batch size of 256 compared to using a batch size of 1; and by about 22 times for lattice-free MMI using a trigram phone language model for the denominator.

Index Terms— forward-backward algorithm, connectionist temporal classification, lattice-free maximum mutual information

1. INTRODUCTION

Deep neural networks are typically trained using gradient-based optimization. When training DNN-based acoustic models for automatic speech recognition, sequence-level loss functions, such as connectionist temporal classification (CTC) [1], maximum mutual information (MMI) [2] and minimum Bayes risk (MBR) [3, 4, 5], are typically used. The forward-backward algorithm plays an important role in computing the gradients of these losses with respect to the model parameters. For CTC, forward-backward is used to compute the soft alignment between the target label sequence and the input logit sequence, which is necessary to compute the CTC loss and its gradients with respect to the logits. For sequence training from lattices, the forward-backward algorithm is used to compute the necessary statistics from lattices [2, 6, 7]. For lattice-free MMI training [8, 9], the denominator statistics needed to compute the MMI loss are collected from a phone n-gram language model.

Previously, we have shown that the forward-backward algorithm can be computed recursively as a series of matrix operations [10]. In fact, the entire computation can be viewed as a bi-directional recurrent neural network. Using matrix operations is attractive because an efficient implementation can be achieved by leveraging existing numerical computation software, such as TensorFlow [11], that already has an optimized implementation of matrix operations for CPUs and GPUs. In order to take full advantage of the compute resources, it is important to be able to process multiple sequences simultaneously. In this paper, we extend our previous forward-backward implementation to support batched processing of variable-length sequences and describe how this can be used to compute the derivatives of the CTC and MMI losses with respect to the logits.

The remainder of this paper is organized as follows. Section 2 presents the recursive formulation of the forward-backward algorithm in matrix form and shows that it can be viewed as a form of recurrent neural network. Section 3 extends the formulation for batched computation that supports variable-length sequences within a batch. Section 4 describes how the batched forward-backward computation can be used for CTC and lattice-free MMI training. Section 5 presents experimental results.

2. FORWARD-BACKWARD ALGORITHM

A finite state model, such as a hidden Markov model (HMM) [12] and an n-gram language model [13], is commonly used to represent sequences. A finite state model comprises a collection of arcs. Each arc defines a transition from one state to another, as well as the transition probabilities, an input label and an output label (for a transducer).

To simplify the implementation of the forward-backward computation in matrix form, we consider a specific form of a finite state machine, where the input and output labels are defined on the states (instead of the arcs), such that each state is uniquely associated with only one input label and one output label[1]. This makes in easy to convert between the label and state probabilities via a sparse matrix multiplication operation (see Fig. 1). Therefore, the parameters associated with a finite state machine is given by $\mathcal{F} = \{\boldsymbol{A}, \boldsymbol{\pi}_I, \boldsymbol{\pi}_F, \boldsymbol{M}_I, \boldsymbol{M}_O\}$, where

- $\boldsymbol{A}$ is a $S \times S$ state transition probability matrix;
- $\boldsymbol{\pi}_I$ is a $S \times 1$ initial state probability vector;
- $\boldsymbol{\pi}_F$ is a $S \times 1$ final state probability vector;
- $\boldsymbol{M}_I$ is a $S \times N_I$ input label to state mapping matrix; and
- $\boldsymbol{M}_O$ is a $N_O \times S$ state to output label mapping matrix.

S, N_I and N_O are the number of states, input and output labels, respectively. The (i, j)-th element of the mapping matrices, $\boldsymbol{M}_I$ and $\boldsymbol{M}_O$, are defined as:

$$m_I[i,j] = \begin{cases} 1 & \text{if state } i \text{ maps to input label } j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$m_O[i,j] = \begin{cases} 1 & \text{if state } j \text{ maps to output label } i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

[1]In other words, all the incoming arcs have the same input and output labels. All finite state machines can be transformed into this representation, at the expense of a larger number of states.

Note that every row of $\boldsymbol{M}_I$ is a one-hot vector, due to the requirement that each state is associated with only one input label. Likewise, each column of $\boldsymbol{M}_O$ is a one-hot vector. In the case where the input and output labels are the same, $\boldsymbol{M}_I = \boldsymbol{M}_O^\top$.

The likelihood of the model generating the training data, $\boldsymbol{X}_1^T = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T\}$, is given by

$$P_\theta\left(\boldsymbol{X}_1^T\right) = \underbrace{\boldsymbol{\pi}_F^\top \boldsymbol{B}_T \boldsymbol{A} \ldots \boldsymbol{B}_{t+1}\boldsymbol{A}}_{\boldsymbol{\beta}_t^\top} \underbrace{\boldsymbol{B}_t \boldsymbol{A} \ldots \boldsymbol{B}_1 \boldsymbol{A} \boldsymbol{\pi}_I}_{\boldsymbol{\alpha}_t} \quad (3)$$

where $\boldsymbol{x}_t$ is the acoustic feature at frame t and $\boldsymbol{B}_t$ is a diagonal matrix, whose leading diagonal elements represent the state observation probabilities at time t, $\boldsymbol{b}_t = \boldsymbol{M}_I \tilde{\boldsymbol{b}}_t$, where $\tilde{\boldsymbol{b}}_t$ is generated by an acoustic model with parameters, θ[2]. $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ are the forward and backward probability vectors at time t, respectively. These probabilities can be computed recursively as:

$$\boldsymbol{\alpha}_t = \boldsymbol{B}_t \boldsymbol{A} \boldsymbol{\alpha}_{t-1} = \boldsymbol{b}_t \odot \boldsymbol{A}\boldsymbol{\alpha}_{t-1} \quad (4)$$
$$\boldsymbol{\beta}_t = \boldsymbol{A}^\top \boldsymbol{B}_{t+1} \boldsymbol{\beta}_{t+1} = \boldsymbol{A}^\top \left(\boldsymbol{b}_{t+1} \odot \boldsymbol{\beta}_{t+1}\right) \quad (5)$$

The initial conditions are given by $\boldsymbol{\alpha}_0 = \boldsymbol{\pi}_I$ and $\boldsymbol{\beta}_T = \boldsymbol{\pi}_F$. Since $\boldsymbol{B}_t$ is a diagonal matrix, it is simpler to use an element-wise multiplication, '$\odot$', operation [10]. The forward and backward probabilities are useful for computing the derivative of the log likelihood with respect to the observation log probabilities:

$$\frac{\log P_\theta\left(\boldsymbol{X}_1^T\right)}{\log \boldsymbol{b}_t} = \tilde{\boldsymbol{\gamma}}_t = \boldsymbol{M}_O \boldsymbol{\gamma}_t \quad (6)$$

where

$$\boldsymbol{\gamma}_t = \frac{\boldsymbol{\beta}_t \odot \boldsymbol{\alpha}_t}{\boldsymbol{\beta}_t^\top \boldsymbol{\alpha}_t} \quad (7)$$

Given the recursive nature of the forward-backward algorithm over time, the computation of $\boldsymbol{\gamma}_t$ can be viewed as a bi-directional recurrent neural network (RNN), where $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ represent the RNN states for the forward and backward recurrence, respectively; and the weights of the RNN are given by $\boldsymbol{A}$, $\boldsymbol{M}_I$ and $\boldsymbol{M}_O$. The RNN architecture that represents a forward-backward computation is depicted in Fig. 1. Therefore, any existing framework that supports an RNN architecture can be easily adapted to compute the forward backward algorithm. In our case, we use an implementation similar to the `dynamic_rnn`[3] operation in TensorFlow [11] to handle full unrolling over the entire sequence.

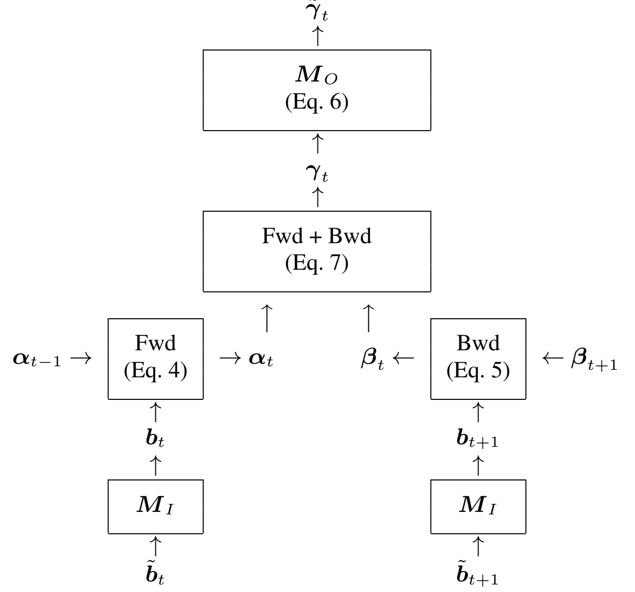


Fig. 1: Forward-backward as a bi-directional recurrent neural network.

3. BATCHED FORWARD-BACKWARD

To fully exploit the multi-core computing architecture (especially for GPUs), a common strategy is to process input sequences in batches, such that a large number of operations can occur in parallel. In this section, we will explain how the forward-backward implementation described in the previous section can be extended to support batched processing of variable-length sequences.

The expressions in Eqn 4 and 5 can be easily extended to process multiple sequences of equal length by simply stacking the probability vectors to form matrices:

$$\bar{\boldsymbol{\alpha}}_t = \bar{\boldsymbol{b}}_t \odot \boldsymbol{A}\bar{\boldsymbol{\alpha}}_{t-1} \quad (8)$$
$$\bar{\boldsymbol{\beta}}_t = \boldsymbol{A}^\top \left(\bar{\boldsymbol{b}}_{t+1} \odot \bar{\boldsymbol{\beta}}_{t+1}\right) \quad (9)$$

where $\bar{\boldsymbol{\alpha}}_t$, $\bar{\boldsymbol{\beta}}_t$ and $\bar{\boldsymbol{b}}_t$ are $S \times B$ matrices given by

$$\bar{\boldsymbol{\alpha}}_t = \left[\begin{array}{cccc}\boldsymbol{\alpha}_t^1 & \boldsymbol{\alpha}_t^2 & \ldots & \boldsymbol{\alpha}_t^B\end{array}\right] \quad (10)$$
$$\bar{\boldsymbol{\beta}}_t = \left[\begin{array}{cccc}\boldsymbol{\beta}_t^1 & \boldsymbol{\beta}_t^2 & \ldots & \boldsymbol{\beta}_t^B\end{array}\right] \quad (11)$$
$$\bar{\boldsymbol{b}}_t = \left[\begin{array}{cccc}\boldsymbol{b}_t^1 & \boldsymbol{b}_t^2 & \ldots & \boldsymbol{b}_t^B\end{array}\right] \quad (12)$$

B is the batch size. $\boldsymbol{\alpha}_t^b$, $\boldsymbol{\beta}_t^b$ and $\boldsymbol{b}_t^b$ denote the forward, backward and state observation probabilities for the b-th sequence, respectively.

A typical approach to handle sequences with different lengths is to pad the shorter sequences with zeros at the end to match the longest sequence in the batch. The extra elements in the outputs as a result of padding are later ignored or discarded to yield the correct results. By choosing an appropriate batch size, the benefit from batched processing can outweigh the unnecessary computation for the padded inputs. The amount of padding required can be minimized through bucketing[4], where sequences of similar lengths are batched together. Bucketing is not used for the experiments in this paper.

We simplify the discussion below by considering padding a sequence within a batch of length T with Δ additional frames. Each sequence within a batch will have a different T and Δ such that $T + \Delta$ is the same for all the sequences within a batch. Consider padding a state observation probability sequence of length T, $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_T\}$ by zeros to the end:

$$\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_T, \underbrace{\boldsymbol{0}_1, \ldots, \boldsymbol{0}_\Delta}_{\text{padded}}\} \quad (13)$$

Applying the forward recursion in Eq. 4 to the padded input sequence will yield the following forward probability sequence:

$$\{\underbrace{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_T}_{\text{desired}}, \boldsymbol{\alpha}_1', \ldots, \boldsymbol{\alpha}_\Delta'\} \quad (14)$$

[2]The dependence on θ is implied for $\boldsymbol{b}_t$ and $\tilde{\boldsymbol{b}}_t$ and dropped for clarity.

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

[4]https://www.tensorflow.org/versions/r0.12/api_docs/python/contrib.training/bucketing

The first T vectors correspond to the same forward probabilities as computed by applying the forward algorithm to the original input sequence.

However, for the backward recursion, the input sequence has to be padded in the *front*, such that:

$$\{\underbrace{\mathbf{0}_1,\ \ldots,\ \mathbf{0}_\Delta}_{\text{padded}},\ \boldsymbol{b}_1,\ \ldots,\ \boldsymbol{b}_T\} \tag{15}$$

Therefore, applying the backward algorithm to the padded input sequence will yield:

$$\{\boldsymbol{\beta}'_1,\ \ldots,\ \boldsymbol{\beta}'_\Delta,\ \underbrace{\boldsymbol{\beta}_1,\ \ldots,\ \boldsymbol{\beta}_T}_{\text{desired}}\} \tag{16}$$

The last T vectors correspond to the same backward probabilities as computed by applying the backward algorithm to the original input sequence.

By reordering the backward probability sequence such that the desired outputs are aligned to the front, element-wise multiplication and sum-to-one normalization in Eq. 7 can be easily applied:

$$\begin{array}{ccccccc} \boldsymbol{\alpha}_1 & \boldsymbol{\alpha}_2 & \ldots & \boldsymbol{\alpha}_T & \boldsymbol{\alpha}'_1 & \ldots & \boldsymbol{\alpha}'_\Delta \\ \boldsymbol{\beta}_1 & \boldsymbol{\beta}_2 & \ldots & \boldsymbol{\beta}_T & \boldsymbol{\beta}'_1 & \ldots & \boldsymbol{\beta}'_\Delta \\ \downarrow & \downarrow & & \downarrow & \downarrow & & \downarrow \\ \boldsymbol{\gamma}_1 & \boldsymbol{\gamma}_2 & \ldots & \boldsymbol{\gamma}_T & \boldsymbol{\gamma}'_1 & \ldots & \boldsymbol{\gamma}'_\Delta \end{array} \tag{17}$$

The unwanted results, $\{\boldsymbol{\gamma}'_1,\ \ldots,\ \boldsymbol{\gamma}'_\Delta\}$, are then discarded by setting them to zero.

4. CTC AND MMI

We now describe how the batched forward-backward algorithm presented in the previous section can be used to compute the derivatives of the CTC [1] and MMI [2] losses with respect to the logits. These derivative are given by:

$$\frac{\partial \mathcal{L}_{\texttt{CTC}}}{\partial \boldsymbol{h}_t} = \tilde{\boldsymbol{\gamma}}_t^n - \tilde{\boldsymbol{b}}_t \tag{18}$$

$$\frac{\partial \mathcal{L}_{\texttt{MMI}}}{\partial \boldsymbol{h}_t} = \tilde{\boldsymbol{\gamma}}_t^n - \tilde{\boldsymbol{\gamma}}_t^d \tag{19}$$

where $\tilde{\boldsymbol{\gamma}}_t^n$ and $\tilde{\boldsymbol{\gamma}}_t^d$ are the forward-backward probabilities computed over the numerator and denominator finite state models, respectively; $\tilde{\boldsymbol{b}}_t = \mathrm{softmax}(\boldsymbol{h}_t)$.

In automatic speech recognition, the input feature length is longer than the label sequence length. A Hidden Markov model (HMM) [12] is typically used to model the duration of the labels. The numerator and denominator finite state models are given by the composition of the HMM duration model and a label sequence model.

Let the parameters of the duration and sequence finite state models be given by $\mathcal{F}^{\texttt{dur}}$ and $\mathcal{F}^{\texttt{seq}}$, respectively:

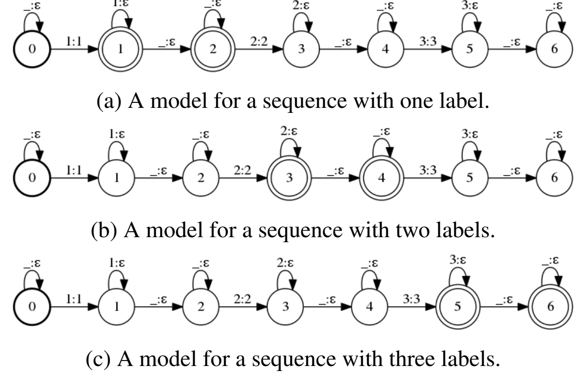
$$\mathcal{F}^{\texttt{dur}} = \{\boldsymbol{A}^{\texttt{dur}}, \boldsymbol{\pi}_I^{\texttt{dur}}, \boldsymbol{\pi}_F^{\texttt{dur}}, \emptyset, \emptyset\} \tag{20}$$

$$\mathcal{F}^{\texttt{seq}} = \{\boldsymbol{A}^{\texttt{seq}}, \boldsymbol{\pi}_I^{\texttt{seq}}, \boldsymbol{\pi}_F^{\texttt{seq}}, \boldsymbol{M}_I^{\texttt{seq}}, \boldsymbol{M}_O^{\texttt{seq}}\} \tag{21}$$

The corresponding number of states for these models are given by $S_{\texttt{dur}}$ and $S_{\texttt{seq}}$, respectively. Assuming that the same duration model is applied to all the labels in the sequence model, the finite state machine after composing $\mathcal{F}^{\texttt{dur}}$ and $\mathcal{F}^{\texttt{seq}}$ is given by $\mathcal{F} = \{\boldsymbol{A}, \boldsymbol{\pi}_I, \boldsymbol{\pi}_F, \boldsymbol{M}_I, \boldsymbol{M}_O\}$:

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{A}^{\texttt{dur}} \otimes \boldsymbol{I}_{S_{\texttt{seq}}} + \Big(\boldsymbol{k}_{S_{\texttt{dur}}} \otimes \boldsymbol{A}^{\texttt{seq}}\Big)\Big(\boldsymbol{I}_{S_{\texttt{seq}}} \otimes \boldsymbol{\pi}_F^{\texttt{dur}\top}\Big) \\ \boldsymbol{\pi}_I &= \boldsymbol{\pi}_I^{\texttt{dur}} \otimes \boldsymbol{\pi}_I^{\texttt{seq}} \\ \boldsymbol{\pi}_F &= \boldsymbol{\pi}_F^{\texttt{dur}} \otimes \boldsymbol{\pi}_F^{\texttt{seq}} \end{aligned}$$

where $\otimes$ is a Kronecker product operator. $\boldsymbol{I}_{S_{\texttt{seq}}}$ is a $S_{\texttt{seq}} \times S_{\texttt{seq}}$ identity matrix and $\boldsymbol{k}_{S_{\texttt{dur}}}$ is a $S_{\texttt{dur}} \times 1$ vector given by $[1, 0, \ldots, 0]^\top$. $\boldsymbol{A}$ is a summation of two terms. The first term corresponds to the within-label state transitions and the second term corresponds to the between-label transitions. The composed model has $S = S_{\texttt{dur}} \times S_{\texttt{seq}}$ number of states. The mapping matrices can be defined to appropriately map the states to the labels. For example, for CTC, the first $S_{\texttt{seq}}$ states will use the mapping from $\boldsymbol{M}_I^{\texttt{seq}}$ and $\boldsymbol{M}_O^{\texttt{seq}}$, while the remaining states will be mapped to a 'blank' label [1].



(a) A model for a sequence with one label.

(b) A model for a sequence with two labels.

(c) A model for a sequence with three labels.

Fig. 2: Representing sequences of different lengths with finite state models sharing the same states and state transitions.

For lattice-free MMI [8, 9], $\tilde{\boldsymbol{\gamma}}_t^d$ is computed using a phone n-gram model as the denominator for all the sequences. Since the same n-gram model is used for all the utterances, the forward-backward algorithm can be computed efficiently on GPUs without excessive host-device data transfer. In this case, the batched forward-backward computation described in the previous section can be applied directly. On the other hand, since each sequence has a different numerator model, the batched version of the forward-backward computation has to be modified in order to compute $\tilde{\boldsymbol{\gamma}}_t^n$. Instead of explicitly representing a different transition matrix for each sequence, it turns out that, for the numerator, it is possible to use the same state transition matrix, $\boldsymbol{A}$, for all the sequences of different lengths. This is achieved via *state-padding*, a concept similar to input padding. The finite state model for a shorter sequence can be padded with additional states and state transitions such that resulting model represents a longer sequence. Therefore, a common model that represents the longest sequence in a batch can be used for all the sequences, provided that sequence-dependent final states are used to appropriately model the sequence lengths. This allows Eq. 8 and 9 to be used. In Fig. 2, a CTC model representing a sequence of three labels is used to model sequences of length 1, 2, and 3 in Fig. 2a, 2b and 2c, respectively. For example, in Fig. 2a, states 1 and 2 are the final states, making states 3, 4, 5 and 6 unreachable. The model effectively represents a one-label sequence. Similarly, setting states 4 and 5 to be the final states yields a model for a two-label sequence (Fig. 2b). Note that the mapping matrices, $\boldsymbol{M}_I$ and $\boldsymbol{M}_O$, have to be represented separately for each sequence.

4.1. Coping with Numerical Stability

Computing the forward-backward algorithm using matrix operations in the probability domain, as described in the previous sections, may lead to numerical instability when the length of input sequence is long. To mitigate this problem, we use the scaling technique proposed in [12] to normalize the forward-backward probabilities at each time to sum to one. However, this is not sufficient to prevent numerical underflow (see results in Section 5.4). Another solution to circumvent this problem is to perform the computation in the log domains. Eq. 8 and 9 can be expressed in the log domain as follows:

$$\log \bar{\boldsymbol{\alpha}}_t = \log \bar{\boldsymbol{b}}_t + \rho\left(\log \boldsymbol{A}, \log \bar{\boldsymbol{\alpha}}_{t-1}\right) \quad (22)$$

$$\log \bar{\boldsymbol{\beta}}_t = \rho\left(\log \boldsymbol{A}^\top, \log \bar{\boldsymbol{b}}_{t+1} + \log \bar{\boldsymbol{\beta}}_{t+1}\right) \quad (23)$$

where $\rho(\boldsymbol{X}, \boldsymbol{Y})$ is a function that computes the equivalent of $\log(\exp(\boldsymbol{X}) + \exp(\boldsymbol{Y}))$[5].

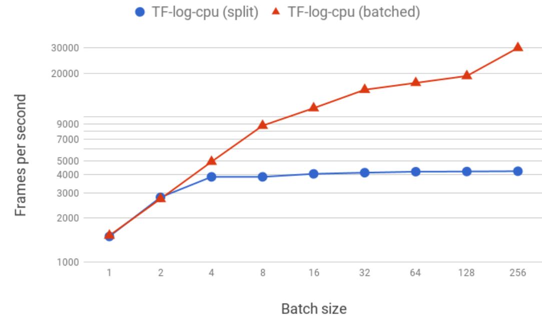
5. EXPERIMENTAL RESULTS

In this section, we present benchmark results to evaluate the effectiveness of the proposed batched forward-backward computation, using CTC [1] and lattice-free MMI (LF-MMI) [8] as example applications. First, we compare the speedup from using the batched computation for both CTC and LF-MMI. Next, we investigate the stability of computing the CTC loss and gradients in the linear and log domain. All the benchmark experiments for the losses and gradients are run using TensorFlow. CPU experiments run on a machine with 12 Intel Xeon CPU processors at 3.5 GHz; GPU experiments run on a machine with an NVIDIA Tesla K20m GPU. The batched forward-backward algorithm is implemented using native TensorFlow ops, which already have support for GPUs, without requiring custom kernels. We benchmark the effectiveness of the batched forward-backward algorithm by using the computation speed (in terms of frames per second, fps) as the evaluation metric to benchmark our implementations, measured as the median over 10 runs. We also evaluate the impact of the batched computation on the overall speed of training a CTC acoustic model.

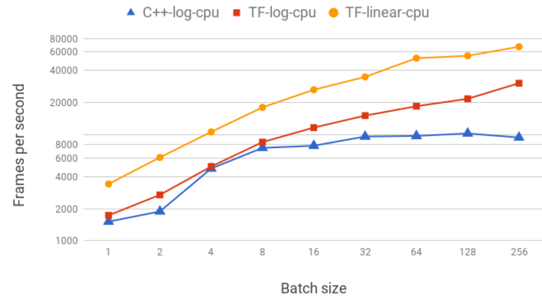
5.1. Benchmark for CTC

First, we benchmark the effectiveness of the batched forward-backward algorithm by the speed of computing the CTC loss and gradients, comparing both the linear- and log-domain implementations running on CPU and GPU. We also compare these implementations with a native C++ implementation. For the benchmark, we use randomly generated label sequences of length 20 and input sequences of 200 frames. We use 42 symbols (logits). Fig. 3a compares the proposed batched computation with a naive approach where the sequences are split and processed individually. Both computations are performed in the log-domain on CPU. TensorFlow internally optimizes the computation by parallelizing the computation of portions of the compute graph that are independent. The naive approach (split) can only take advantage of this 'out-of-the-box' optimization up to a batch size of 4, beyond which the speed saturates at about 4000 fps. On the other hand, we continue to observe speedups from the proposed batched computation as we increase the batch size to 256, where the final speed is about 30,000 fps (about 7 times faster than the naive approach).

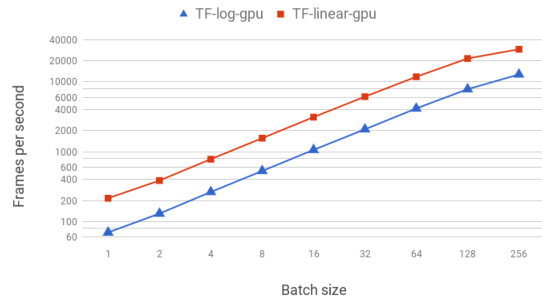
[5] This is natively supported using the `tf.logsumexp` operator.



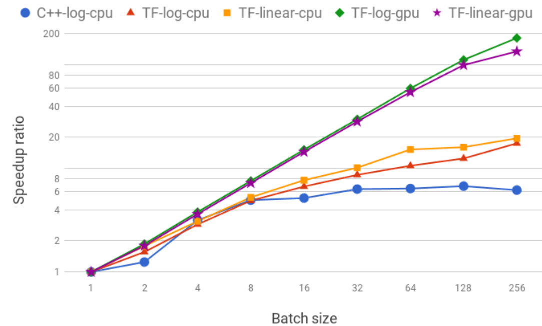
(a) Frames per second (fps) – split vs. batched.



(b) Frames per second (fps) – CPU.



(c) Frames per second (fps) – GPU.



(d) Speedup ratio.

Fig. 3: Speed (fps) and speedup ratio for CTC loss and gradient using batched computation (42 symbols; label sequence length = 20; input sequence length = 200).

Fig 3b and 3c show the frames per second (fps) results on CPU and GPU, respectively. The graphs show that the speed of the TensorFlow implementations increases almost linearly (both the horizontal and vertical axes are in log scales) with the batch size. Note that the linear-domain computation is almost twice as fast as the log-domain computation. Since the C++ implementation of the forward-backward algorithm does not have parallelism support built in, we run each sequence in a separate thread. The benefit from multi-threading saturates between batch sizes of 8 and 16. This roughly corresponds to the number of CPUs available. Note that the C++ implementation (Fig. 3b) is faster than the split version of the TensorFlow implementation (Fig. 3a). This may be due to the overhead of running multiple `dynamic_rnn` unrolling in parallel.

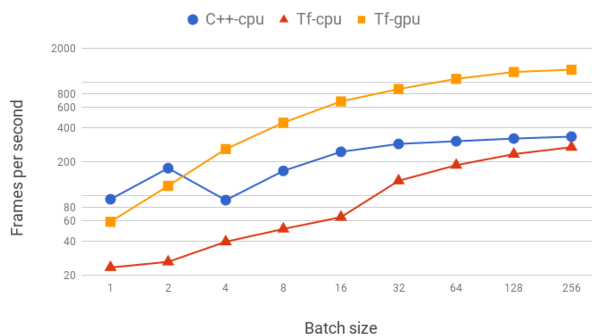
Since it is not meaningful to compare the results from CPU and GPU runs directly, we also plotted the speedup ratios in Fig. 3d to compare all the runs. Speedup ratio is defined as the ratio of the frames per second achieved using a given batch size over that using batch size of one, for each respective setting. It is clear, from the simulation results in Fig. 3d, the proposed batched computation using TensorFlow achieved larger speedups compared to the native C++ implementation. Moreover, when run on GPU, the TensorFlow implementations continue to benefit from batching, up to batch size of 256, where the linear- and log-domain implementations achieved about 135 and 183 times speedup, respectively.

5.2. Benchmark for LF-MMI

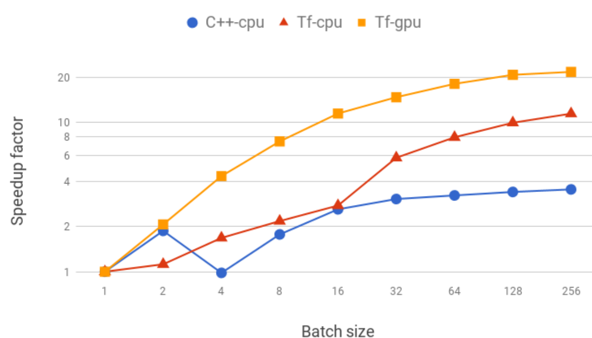
Next, we perform a similar benchmark for LF-MMI. We used log-domain computation for the numerator and linear-domain computation for the denominator (similar to the setup in [8]). The denominator is represented by a trigram phone language model with 42 symbols, 1,674 states and 68,634 transitions. This language model was trained using the transcript of approximately 20,000 hours of anonymized voice-search data. The benchmark results are shown in Fig. 4a and 4b. Due to the more expensive computation of the denominator, the speed does not increase quite as much as compared to CTC when batch size is increased. It is worth noting that the performance of the sparse matrix multiplication operation in TensorFlow is much less efficient when run on CPU compared to GPU, as reported in our previous work [10]. However, it does benefit more from batching compared to the native C++ implementation. As shown in 4b, the TensorFlow implementation achieved speedup ratios of 11 and 22 on CPU and GPU, using a batch size of 256.

5.3. Benchmark for CTC acoustic model

We now present benchmark results when training an acoustic model (AM) using CTC loss. As input, the AM uses 80 dimensional log Mel spectrograms appended with deltas and double deltas, computed every 10 msec. The model has 2 convolutional layers, each with 32 feature maps and convolutional kernels of size (3, 3). After each convolutioanl layer, the outputs are subsampled by a factor of 2 along the time and frequency axes. The convolutional layers are followed by 3 layers of bidirectoinal LSTMs [14, 15], each with 256 units, and a softmax layer with 256 classes corresponding to grapheme-based targets (inlcuding the CTC-blank). This model was trained on the Wall Street Journal corpus for 1000 steps. Since utterances are of varying lengths, we used the bucketing feature in TensorFlow, grouping utterances with less than 1024 frames into one bucket and the rest into another. The results, shown in Table 1, are the average number of utterances processed by the model per second, when run on a machine with 32 Intel Xeon CPUs at 2.6 GHz and a Quadro M2000



(a) Frames per second (fps).



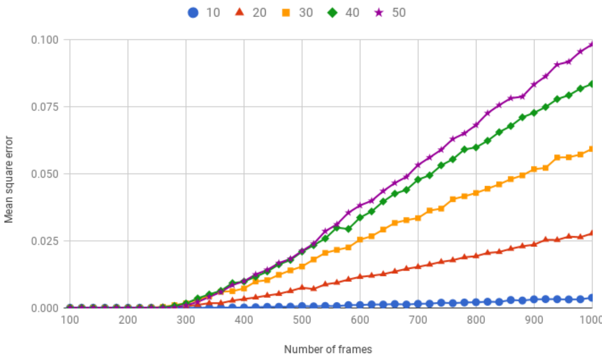
(b) Speedup ratio.

Fig. 4: Speed (fps) and speedup ratio for MMI loss and gradient using batched computation (42 symbols; label sequence length = 20; input sequence length = 200; 3-gram phone language model).

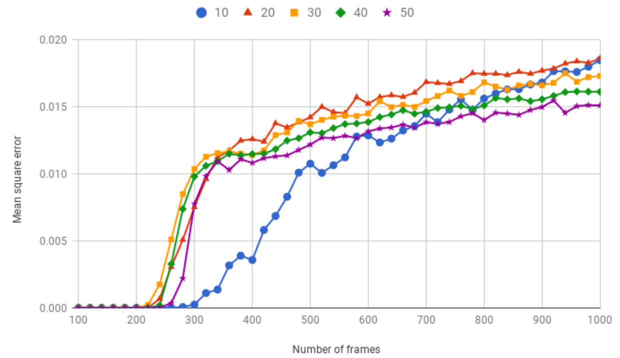
GPU. To show the advantages of the proposed batching mechanism, we compare with the 'split' approach mentioned before, where the loss and gradients are computed by running CTC separately on each uttearnce in the batch (TF-log-gpu (split) in Table 1). Note that all of the intermediate layers are still batched, and only the loss is computed via splitting. As expected, the overall gains are smaller compared to the results presented in previous sections because computation is dominated by the lower layers. Nonetheless, as the batch size increases, speed-up by batching increases at a faster rate; when batch size is 32, batching is twice as fast.

Table 1: Speed (utterances/sec) and speedup ratio of batching when training an acoustic model.

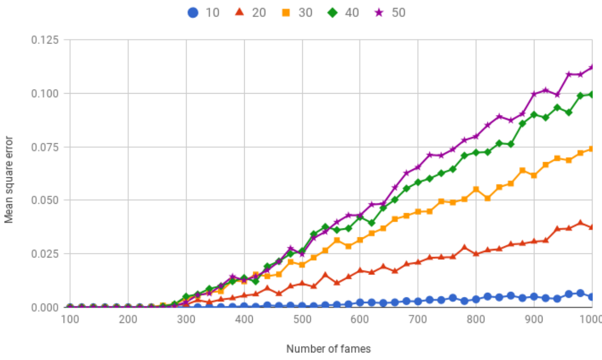
Batch size	TF-log-gpu (split)	TF-log-gpu (batched)	Speedup ratio
1	0.6	0.6	1.0
2	1.0	1.0	1.0
4	1.6	1.8	1.1
8	2.5	3.2	1.3
16	3.8	5.3	1.4
32	4.1	7.6	1.9



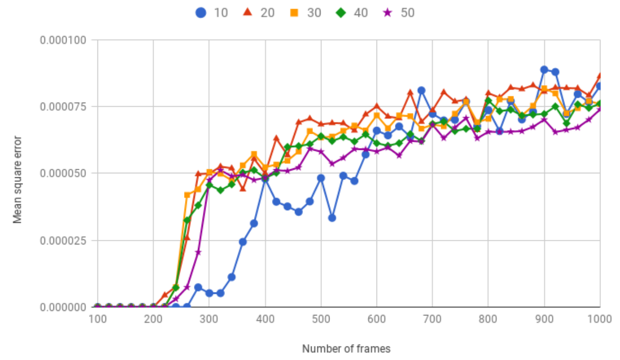
(a) Mean square error of CTC loss with 40 symbols.



(b) Mean square error of CTC gradients with 40 symbols.



(c) Mean square error of CTC loss with 9,000 symbols.



(d) Mean square error of CTC gradients with 9,000 symbols.

Fig. 5: Comparison of the numerical stability of the CTC implementation in linear vs. log domain, with different number of symbols, label sequence length and number of input frames.

5.4. Numerical Stability for CTC

Although computing the forward-backward algorithm in the linear domain may be simpler and faster, it may suffer from numerical instability issue for long sequences. We ran simulations to study the effect of sequence lengths on the numerical stability issues for CTC. Fig. 5 shows the simulation results. The simulations are set up as follows. We considered both context-independent (number of symbols = 40) and context-dependent (number of symbols = 9,000) phone models. For both scenarios, we compute the mean square errors between the linear- and log-domain computations for the CTC loss and gradients. We varied the input sequence length between 100 and 1,000 frames; and the target label sequence length between 10 and 50. Overall, we observe similar trends for both scenarios; the number of symbols does not make much difference. For the CTC loss computation, as shown in Fig. 5a and 5c, the linear version does not suffer from numerical stability issue for short sequences (input length less than about 250 frames). Beyond that, the mean square error starts to diverge. The divergence is larger for longer label sequence (more states in the finite state model). Similarly, for the gradients (Fig. 5b and 5d), the linear- and log-domain computations are similar for input sequence length smaller than 200 frames. Therefore, for input sequences shorter than 200 frames, it is possible to use the linear-domain computation for better speed performance (as previously shown in Sec 5.1).

6. CONCLUSIONS

This paper has proposed an algorithm to simultaneously compute the forward-backward probabilities for multiple sequences of different lengths. This can be easily achieved by extending our previous implementation using matrix operations. By appropriately padding the inputs and realigning the forward and backward probabilities, it is possible to process multiple sequences of different lengths in batches. The proposed algorithm has been successfully applied to compute the derivatives of the CTC and lattice-free MMI losses with respect to the logits. For long sequences, it is necessary to compute the forward-backward probabilities in the log domain to avoid numerical instability issue. Empirical simulation results show that the proposed method is effective in improving the throughput by fully utilizing the available compute resources.

7. REFERENCES

[1] Alex Graves, Santiago Fernandez, Faustino Gomez, and Jurgen Schmidhuber, “Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks,” in *International Conference on Maching Learning*, 2006, pp. 369–376.

[2] V Valtchev, JJ Odell, Philip C Woodland, and Steve J Young,

"MMIE training of large vocabulary recognition systems," *Speech Communication*, vol. 22, no. 4, pp. 303–314, 1997.

[3] Janez Kaiser, Bogomir Horvat, and Zdravko Kačič, "Overall risk criterion estimation of hidden Markov model parameters," *Speech Communication*, vol. 38, no. 3, pp. 383–398, 2002.

[4] Matthew Gibson and Thomas Hain, "Hypothesis spaces for minimum Bayes risk training in large vocabulary speech recognition.," in *Interspeech*, 2006, vol. 6, pp. 2406–2409.

[5] Daniel Povey and Brian Kingsbury, "Evaluation of proposed modifications to MPE for large scale discriminative training," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*. IEEE, 2007, vol. 4, pp. IV–321.

[6] Daniel Povey and Philip C Woodland, "Minimum phone error and I-smoothing for improved discriminative training," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*. IEEE, 2002, vol. 1, pp. I–105.

[7] Brian Kingsbury, "Lattice-based optimization of sequence classification criteria for neural-network acoustic modeling," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*. IEEE, 2009, pp. 3761–3764.

[8] Daniel Povey, Vijayaditya Peddinti, Daniel Galvez, Pegah Ghahrmani, Vimal Manohar, Xingyu Na, Yiming Wang, and Sanjeev Khudanpur, "Purely sequence-trained neural networks for ASR based on lattice-free MMI," in *Interspeech*, 2016.

[9] W Xiong, J Droppo, X Huang, F Seide, M Seltzer, A Stolcke, D Yu, and G Zweig, "The Microsoft 2016 conversational speech recognition system," *arXiv preprint arXiv:1609.03528*, 2016.

[10] Khe Chai Sim and Arun Narayanan, "An efficient phone n-gram forward-backward computation using dense matrix multiplication," in *Interspeech*, 2017.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[12] Lawrence R Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[13] Reinhard Kneser and Hermann Ney, "Improved backing-off for m-gram language modeling," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*. IEEE, 1995, vol. 1, pp. 181–184.

[14] Sepp Hochreiter and Jürgen Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] Mike Schuster and Kuldip K Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.