End-to-End Training of Acoustic Models for Large Vocabulary Continuous Speech Recognition with TensorFlow

Ehsan Variani, Tom Bagby, Erik McDermott, Michiel Bacchiani

Google Inc, Mountain View, CA, USA

{variani, tombagby, erikmcd, michiel}@google.com

Abstract

This article discusses strategies for end-to-end training of state-of-the-art acoustic models for Large Vocabulary Continuous Speech Recognition (LVCSR), with the goal of leveraging TensorFlow components so as to make efficient use of large-scale training sets, large model sizes, and high-speed computation units such as Graphical Processing Units (GPUs). Benchmarks are presented that evaluate the efficiency of different approaches to batching of training data, unrolling of recurrent acoustic models, and device placement of TensorFlow variables and operations. An overall training architecture developed in light of those findings is then described. The approach makes it possible to take advantage of both data parallelism and high speed computation on GPU for state-of-the-art sequence training of acoustic models. The effectiveness of the design is evaluated for different training schemes and model sizes, on a 15,000 hour Voice Search task.

Index Terms: speech recognition, tensorflow

1. Introduction

In recent years there has been an explosion of research into new acoustic models for LVCSR. Large performance gains have resulted from the shift from Gaussian Mixture Models (GMMs) to feed-forward Deep Neural Networks (DNNs) trained at the frame level [1], and further gains have been obtained from utterance-level sequence training of DNNs [2, 3]. Significant additional gains have resulted from switching from feed-forward DNNs to Recurrent Neural Networks (RNNs), in particular Long Short Term Memory Models (LSTMs), again trained both at the frame level and sequence level [4]. Further architecture exploration has included evaluation of Convolutional Neural Networks [5] and much deeper networks such as Residual Networks [6].

Recent work has also proposed end-to-end *models* for joint sequence-level acoustic and language modeling [7, 8], with corresponding sequence-level training. End-to-end *training*, however, is not unique to these new models; indeed, discriminative sequence training of acoustic models for ASR has a long history [9, 10, 11, 12].

A common thread in the rapid evolution of modeling architectures is increasing model size, and use of larger training sets, creating new challenges for the efficient use of computational resources. There is a clear need for well engineered, carefully designed acoustic model trainers which can leverage both data and model parallelism and high performance computational resources such as Graphical Processing Units (GPUs), while still allowing the use of the complex, non-cohesive decoder or lattice operations necessary for state-of-the-art sequence training, which easily scales up to much larger models and larger training data sets, and that produces state-of-the-art performance.

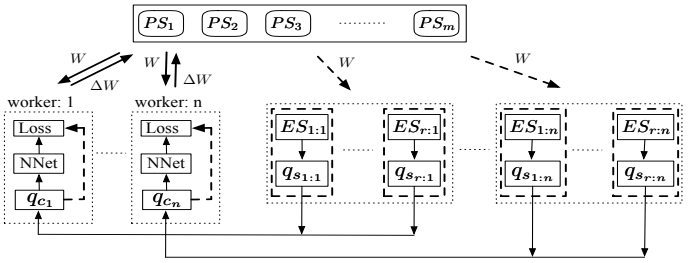


Figure 1: *The end-to-end training system: Parameter Server (PS), Example Server (ES) and workers. The Example Servers pass training examples to server queues q_s which can be called by worker queues q_c. The ES has access to the PS, necessary for sequence training.*

A number of general purpose neural network training packages such as TensorFlow [13] are now available to the research community, as are off-the-shelf fast computation hardware resources such as GPUs. However, speech recogition has its own specific needs. For example, the current state-of-the-art model in speech is the LSTM, which presents unique challenges for efficient computation. The specific choices for unrolling, e.g. full unrolling up to the utterance length [14] vs truncated unrolling [15], optionally with state-saving, and the choice of batching, have significant consequences for training efficiency. In addition to the modeling complexities, there are ASR-specific optimization challenges. In particular, sequence training of acoustic models is a critical step, but the decoder and lattice operations it involves are not easy to implement on GPUs efficiently, in a manner that scales to larger models. This has led to hybrid use of CPU and GPU (placing decoding/lattice operations on the CPU, and computation of local acoustic posterior scores on the GPU) [16], "lattice free" use of a simplified training LM with forward-backward based sequence training on GPUs [17], or biting the bullet on porting complex lattice operations to GPUs [18], among many studies.

In this article we present a comprehensive view of these critical issues in acoustic model trainer design, in light of the functions provided by the TensorFlow platform. We benchmark different strategies for recurrent model unrolling, data batching, and device placement of TensorFlow operations. We discuss how the distribution of training data utterance length can inform these choices. Building on a previous approach to asynchronous sequence training of DNNs and LSTMs [4, 19], we describe an architecture in which the computation of outer and inner derivatives is decoupled between CPU and GPU, while still leveraging data parallelism using e.g. 100s of workers. We then present experimental results evaluating this TensorFlow-based training architecture on a 15,000 hour Voice Search training set, for different model sizes, achieving high final word accuracy in much reduced training times.

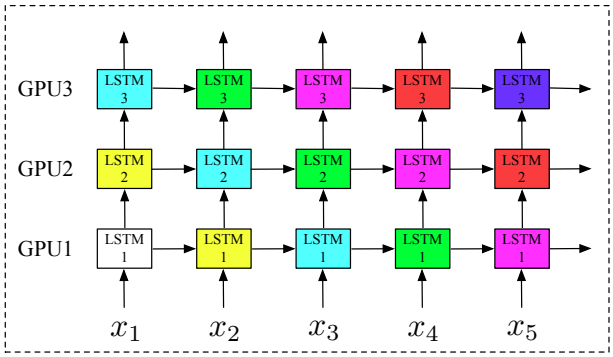


Figure 2: *Model parallelisation across 3 LSTM layers unrolled for 5 time steps. Iterations with same color are performed at same time.*

2. Acoustic Modeling Training with TensorFlow

TensorFlow distinguishes itself from other machine learning platforms such as DistBelief[20] in the way it expresses computation as stateful dataflow graphs. The dataflow computation paradigm and ability to distribute computation across machines gives us a powerful tool for acoustic model training. With careful placement of different parts of this graph onto different machines and devices, we can train state-of-the-art LVCSR systems end-to-end using a mix of GPU and CPU devices.

2.1. Distributed Feature Extraction

Typical ASR front-end consists of a sequence of computations such as Framing, Windowing, Short Term Fourier Transfom, and finally log Mel filtering. For small training sets, feature extraction is normally performed offline, storing features onto disk. State-of-the-art systems today train using thousands of hours of training data, making it impractical to store features on disk. One way to handle this is to represent the front-end as a TensorFlow computation graph and distribute it across different machines along with neural network models and optimization subgraphs. Depending on the complexity of the front-end, this can impose a significant computation load in addition to the computation load needed for each step of forward and backward operation on the training workers. This is certainly the case for multi-style training, in which noise is added to training utterances on-the-fly. Alternatively, one can distribute the front-end subgraph into different sets of machines, as shown in the right side of Figure 1. Each machine, denoted as an Example Server (ES), is a multi-core CPU machine with access to a shard of training data, an optionally for sequence training, access to the parameter servers in order to compute sequence loss outer derivatives. The computation graph is distributed across multi-core machines to perform on-the-fly feature extraction. After feature extraction, each server passes extracted features in a serialized format to a corresponding *queue*. For each worker i, there are r example servers, $ES_{1:i}$, ..., $ES_{1:r}$ extracting features using a shared computation graph. Each example server converts features to `tf.SequenceExample` format and streams the result back to the corresponding training worker using queues, $q_{s_{1:i}}$, ..., $q_{s_{r:i}}$, which produce batches of training data using standard parsing and queueing functions.

2.2. Parallelization methods

Data parallelism uses copies of the model, processing different batches from the training data in parallel. In a standard TensorFlow data parallel training setup, each training replica computes gradients for a mini-batch and then asynchronously applies gradient updates to weights stored in a parameter server.

For model parallelism, we can assign each layer of our stacked LSTM architecture to its own GPU. Figure 2 shows the dependency graph for an RNN. As iteration n of a given layer only depends on iteration $n-1$ of that layer and iteration n of the previous layer, it is possible to start on a layer before finishing the previous layer. Parallelizing data movement between GPUs is handled transparently by TensorFlow op scheduling, and we obtain a speed-up nearly linear in the number of layers. The limitation is that we can only parallelize over distinct parts of the architecture. We can add GPUs for each layer, but cannot speed up further by using multiple GPUs per layer.

2.3. Training Recurrent architecture

TensorFlow provides two functions for unrolling RNNs: `static_rnn` and `dynamic_rnn`. The weights and architecture of the RNN are separated from the mechanism of unrolling them in time. `static_rnn` creates an unrolled graph for a fixed RNN length; a complete subgraph of the RNN operations is repeated for each time step. The limitations of this are excess memory use, slow initial creation of a large graph, and sequence length cannot be longer than the fixed unrolling. However, this simple unrolling does have a side effect of transparently pipelining time steps by the nature of TensorFlow op execution, which can be exploited for easy model parallelism shown in Figure 2. `dynamic_rnn` solves memory and sequence length limitations by using a `tf.While` loop to dynamically unroll the graph when it is executed. That means graph creation is faster and batches can be of variable sequence length. Performance for both is similar, and both can be used with state saving for truncated unrolling.

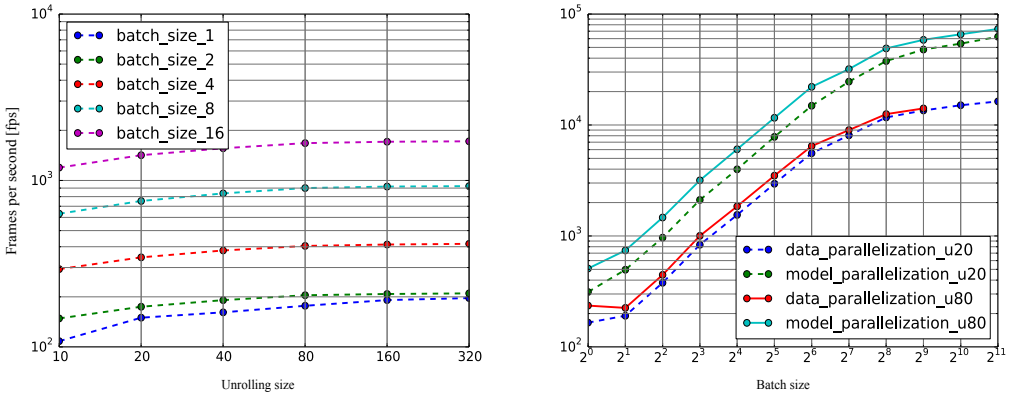
Figure 3 shows the number of frames per second for different batch and unrolling sizes. It can be seen that increasing batch size increases throughput much more than longer unrolling. The fixed unrolling setup can be faster using model parallelization (Figure 2), placing each layer of the model on one GPU. As Figure 3-b shows, this can give up to 4x speed up.

For full unrolling we must handle variable length sequences from the training data to avoid every batch being domintated by a long outlier sequence. This can be handled by bucketing utterances into batches of similar length, which TensorFlow has utility functions to support, `bucket`. Total batch size is still limited by the memory needed to process batches made from the longest sequences in the training distribution.

3. End-to-end training scheme

Cross Entropy (CE) training is performed using the tandem of example servers described earlier, running on CPU, and training workers, running on either CPU or GPU, that compute model updates using the standard CE loss function, calculated with the features and targets provided by the example servers.

Sequence training uses the same tandem of example servers and training workers, but with outer derivatives instead of CE targets, and a simple auxiliary loss function that implements an asynchronous chaining of outer and inner derivatives [19]



a) Frames per second for different batch and unrolling size.

b) Model vs. data parallelization for unrolling of size 20 and 80.

Figure 3: *Benchmark of different training schemes for training 5 layers LSTM with 768 cells per layer using 5 GPUs.*

constituting the total sequence-level loss gradient,

$$\frac{\partial \mathcal{L}\left(\mathbf{X},\theta\right)}{\partial \theta_i} = \sum_{t=1}^{T}\sum_{k}^{N} \frac{\partial \mathcal{L}\left(\mathbf{X},\theta\right)}{\partial a\left(\mathbf{x}_t,k,\theta\right)} \frac{\partial a\left(\mathbf{x}_t,k,\theta\right)}{\partial \theta_i}, \qquad (1)$$

for a sequence $\mathbf{X}$ of T feature vectors $\mathbf{x}_t$, unfolded RNN logits $a\left(\mathbf{x}_t,k,\theta\right)$, for all N network output classes, and all network parameters θ_i [3]. The example servers compute the outer derivatives,

$$w\left(\mathbf{x}_t,k,\theta'\right) = \left.\frac{\partial \mathcal{L}\left(\mathbf{X},\theta\right)}{\partial a\left(\mathbf{x}_t,k,\theta\right)}\right|_{\theta=\theta'}, \qquad (2)$$

for a snapshot of the parameters θ' obtained from the parameter server, and the training workers use the auxiliary function

$$\mathcal{L}_{AUX}\left(\mathbf{X},\theta\right) = \sum_{t=1}^{T}\sum_{k=1}^{N} w\left(\mathbf{x}_t,k,\theta'\right) \log p_\theta\left(\mathbf{x}_t|k\right), \qquad (3)$$

where $p_\theta\left(\mathbf{x}_t|k\right)$ is the network output, i.e. the usual softmax over the logits. It is easy to show that the gradient of this auxiliary function with respect to the live parameters θ then approximates the chaining of outer and inner derivatives in Eq. (1).

In this study, outer and inner derivatives are computed on different machines (example server and training worker), allowing for faster computational throughput, but resulting in a significantly higher degree of asynchrony than in [3, 19]. Tuning the number of workers, queue size, and filtering out outer derivative sequences computed for a θ' deemed overly stale, were found to be important to obtaining good convergence.

The snapshot parameters θ' must be periodically refreshed; this can be implemented in TensorFlow by adding to the inference graph used in the example server sequence training pipeline a set of operations copying live variables on the parameter server to the snapshot variables.

This asynchronous approach to sequence training offers flexibility in the choice of batching and unrolling schemes for RNN training discussed in Section 2.3, as it decouples computation of outer and inner derivatives. As outer derivative computation for the sequence-level loss requires the entire utterance, `dynamic_rnn` can be used. The issues with full unrolling previously discussed are mitigated by the fact that the outer derivatives are computed with just a forward pass. On the inner derivative side, large batches with limited unrollling can leverage the efficiency of GPU-based computation.

Sequence training is performed starting from a CE-trained acoustic model. The sequence training criterion in this study is state-level Minimum Bayes Risk (sMBR) [2, 3].

4. Experiments

Training a recurrent neural network involves the choice of many parameters, including unrolling and batch size. Hardware can impose constraints on the optimal choice of these parameters. The experiments are designed to investigate the effect of these choices on Word Error Rate (WER).

data: The training data consists of about 15000 hours of spontaneous speech from anonymized, human-transcribed Voice Search queries. For noise robustness training, each utterance in the training set is artificially noisified using 100 different styles of noises with varying degrees of noise and reverberation. The test sets are separate anonymized, human-transcribed Voice Search datasets of about 25 hours each. Evaluation is presented on two sets, *clean*, from real Google voice search traffic and *noise*, an artificially noisified version of the clean test set, with the same configuration used for training set noisification.

front-end: The training examples are 480-dimensional frames output by the front end every 30 ms, generated by stacking four frames of 128 log mel filterbank energies extracted from a 32 ms window shifted over the audio samples in 10 ms increments [21]. During CE training, to emulate the effect of the right context, we delay the output state label by 5 frames [22]. Each frame is labeled with one out of 8192 context-dependent output phoneme states; each feature frame and target label are bundled by the example server in `tf.SequenceExample` format. For sequence training, the example server bundles outer derivatives instead of target labels, along with the feature frame and sMBR-related statistics, e.g. the utterance level loss value itself, for summarization in TensorBoard. The outer derivatives are computed over the entire utterance using the model parameters loaded in the example server at the time of processing the utterance, as described in Section 3.

training: ASGD was used for all experiments. For CE training, a ratio of 10 to 1 is used for the number of example servers per worker. For sequence training, a ratio of 30 to 1 is used. All trainer workers are GPUs and use data parallelism. In total, 32 workers were used for CE training; 16 workers were used for sequence training. The parameter servers are multi-core CPU machines; For GPU experiments, 13 parameter servers were used.

Choice of unrolling scheme: The choice of full unrolling vs. truncated unrolling can be evaluated from different perspectives. In Table 1, these are compared in terms of WER performance, convergence time, maximum number of frames per mini-batch and *Average Padding Ratio* (APR), the average number of padded frames per step of training. The model used for comparison was a 5 layer LSTM with 600 cells per layer, denoted as 5xLSTM(600). The fully unrolled model was trained with batch 1, the setting for our best full unrolling system. For truncated unrolling, the model was trained with batch 256 and unrolling of 20. The fully unrolled model was trained using 500 CPU workers and 97 parameter servers. The default GPU setup described above was used for training the truncated model. With respect to WER, both models converge to similar WERs for both clean and noise test sets. The convergence time of the truncated model is significantly shorter than the fully unrolled model with batch size 1, due to the effectiveness of large batches in the GPU setup. Though full unrolling can be sped up using batching (with bucketing of sequences by length), the potential speed up depends on the distribution of sequence length in the training data. When bucketing, the number of frames in each step of SGD varies with the bucket sequence length, affecting step time proportionally. As step time changes, the number of asynchronous steps computed by other replicas changes, introducing more variable parameter staleness. Attempting to control for this, with e.g. variable batch size, introduces more training parameters and complexity.

With truncated unrolling, the maximum number of frames per step is bounded by batch size times unrolling size. The main drawback to batching sequence data is the need for padding each batch to the maximum sequence length in the batch. In our case the average number of padded frames per mini-batch (Table 1) is about one sixth of the mini-batch size. The number of wasted padded frames increases as the unrolling is increased.

For full unrolling, padding depends on bucket size used and depends on the sequence length distribution of the training set. For the training set used for this paper, truncated unrolling was preferred for its simplicity and is used for the rest of experiments presented here.

	WER [%]		Conv.	Max	APR
	clean	noise	[days]	frames	[%]
full	12.07	19.52	28 d	45k	0
truncated	12.10	19.76	6 d	5120	15.6

Table 1: *WER, convergence time and mini-batch metrics (maximum # of frames & Average Padding Ratio (APR)) for full vs truncated unrolling.*

Choice of batching: Truncated unrolling is desirable, as it allows the use of very large batch sizes, which significantly improves the throughput of GPU parallel computation. However, the choice of batch size is not completely independent of the choice of unrolling size.

To examine this behavior, three pairs of batch and unrolling sizes were chosen such that the total number of frames in each mini-batch, $b \times u$, is constant. This allows us to avoid learning rate tuning for each model. Table 2 compares the WER of a 5xLSTM(768) model trained with three batch sizes, 512, 256 and 128 with corresponding unrolling sizes of 10, 20, and 40. The performances are presented for CE. The model trained with unrolling of 20 and 40 performs similarly, while the model trained with larger batch size of 512 and unrolling of 10 shows some performance degradation. This might be due to the fact that the unrolling size is not sufficient for learning longer temporal dependencies, or to optimization issues such as the padding ratio introduced by batching for truncated unrolling. Table 2 presents the mean and standard deviation of the total number of padded frames over training steps. The mean and stddev for unrolling 10 and batch size 512 setup is the largest, compared to the other setups. This means that the number of frames used for learning varies significantly across steps, which might explain the performance degradation.

(b, u)	(512, 10)	(256, 20)	(128, 40)
clean	11.70	11.40	11.45
noise	19.21	18.51	18.56
avg padding ratio	20.6 ± 19.2	15.6 ± 10.0	17.6 ± 7.4

Table 2: *WERs for different batching and unrolling schemes.*

Choice of model parameters: Table 3 summarizes WERs after sequence training for three models with different number of LSTM cells per layer. These models were trained with batch of 256 and unrolling of 20, with learning rate tuned for all models. As discussed in Section 3, outer derivative staleness is an issue. To address that, for each model we zeroed out the 32 most stale examples in each mini-batch.

The model parameters can be chosen to make the best use of the available hardware. In our examples, increasing the layer size together with the batch size allows more efficient use of GPU hardware. In Table 3, two LSTM topologies, one with 600 units per layer and one with 768 units per layer, are trained with same resources; the convergence time for both models is similar. However, the larger model is significantly better in terms of WER. Furthermore, the wider model with 1024 cells per layer shows extra gains but of course this leads to extra training time.

	cross_entropy		sequence_training		Conv.
	clean	noise	clean	noise	[days]
5xLSTM(600)	12.10	19.76	10.43	15.56	6 + 3
5xLSTM(768)	11.40	18.51	10.10	15.01	6.5 + 3
5xLSTM(1024)	11.04	17.35	9.88	14.13	10 + 4

Table 3: *Comparison of WER after sequence training.*

5. Conclusion

This article discussed different approaches to efficient distributed training of RNNs in TensorFlow, with a focus on different strategies for data batching, recurrent model unrolling, and device placement of TensorFlow variables and operations. A training architecture was proposed that allows for flexible design of those strategies, enabling the use of a hybrid distributed CPU/GPU training scheme that extends previous work on asynchronous sequence training. Experimental results on a 15,000 hour Voice Search task show that this training architecture suffers no loss compared to a previous LSTM baseline of the same model size that uses conventional settings (full unrolling, no batching, synchronous sequence training), but in much reduced time, thanks to the GPU-based implementation enabled by the proposed design. Additional gains were obtained with significantly wider models, again in much improved training time.

6. Acknowledgements

The authors would like to thanks all members of Google speech team.

7. References

[1] F. Seide, G. Li, X. Chen, and D. Yu, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Proc. ASRU*, 2011, pp. 24–28.

[2] B. 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.

[3] E. McDermott, G. Heigold, P. J. Moreno, A. W. Senior, and M. Bacchiani, "Asynchronous stochastic optimization for sequence training of deep neural networks: towards big data." in *Interspeech*, 2014, pp. 1224–1228.

[4] H. Sak, O. Vinyals, G. Heigold, A. Senior, E. McDermott, R. Monga, and M. Mao, "Sequence discriminative distributed training of long short-term memory recurrent neural networks," *entropy*, vol. 15, no. 16, pp. 17–18, 2014.

[5] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[7] W. Chan, N. Jaitly, Q. Le, and O. Vinyals, "Listen, attend and spell: A neural network for large vocabulary conversational speech recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 4960–4964.

[8] A. Graves, "Sequence transduction with recurrent neural networks," *CoRR*, vol. abs/1211.3711, 2012.

[9] P. F. Brown, "The acoustic-modeling problem in automatic speech recognition," Ph.D. dissertation, Pittsburgh, PA, USA, 1987.

[10] J. S. Bridle, "Alpha-nets: A recurrent “neural” network architecture with a hidden markov model interpretation," *Speech Commun.*, vol. 9, no. 1, pp. 83–92, Jan. 1990. [Online]. Available: http://dx.doi.org/10.1016/0167-6393(90)90049-F

[11] P. Haffner, "Connectionist speech recognition with a global MMI algorithm," in *EUROSPEECH*. ISCA, 1993.

[12] V. Valtchev, J. Odell, P. C. Woodland, and S. J. Young, "Lattice-based discriminative training for large vocabulary speech recognition," in *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, vol. 2. IEEE, 1996, pp. 605–608.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[14] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[15] R. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Comput.*, vol. 2, no. 4, pp. 490–501, 1990.

[16] K. Veselỳ, A. Ghoshal, L. Burget, and D. Povey, "Sequence-discriminative training of deep neural networks." in *Interspeech*, 2013, pp. 2345–2349.

[17] D. Povey, V. Peddinti, D. Galvez *et al.*, "Purely sequence-trained neural networks for asr based on lattice-free mmi," *Interspeech*, 2016.

[18] H. Su, G. Li, D. Yu, and F. Seide, "Error back propagation for sequence training of context-dependent deep networks for conversational speech transcription," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 6664–6668.

[19] G. Heigold, E. McDermott, V. Vanhoucke, A. Senior, and M. Bacchiani, "Asynchronous stochastic optimization for sequence training of deep neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 5587–5591.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[21] G. Pundak and T. N. Sainath, "Lower frame rate neural network acoustic models," *Interspeech 2016*, pp. 22–26, 2016.

[22] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling." in *Interspeech*, 2014, pp. 338–342.