Deep Speech 2: End-to-End Speech Recognition in English and Mandarin

*Silicon Valley AI Lab (SVAIL)**

**Presenters: Jesse Engel, Shubho Sengupta*

Abstract

We demonstrate a generic speech engine that handles a broad range of scenarios without needing to resort to domain-specific optimizations.

We perform a focused search through model architectures finding deep recurrent nets with multiple layers of 2D convolution and layer-to-layer batch normalization to perform best.

In many cases, our system is competitive with the transcription performance of human workers when benchmarked on standard datasets.

This approach also adapts easily to multiple languages, which we illustrate by applying essentially the same system to both Mandarin and English speech.

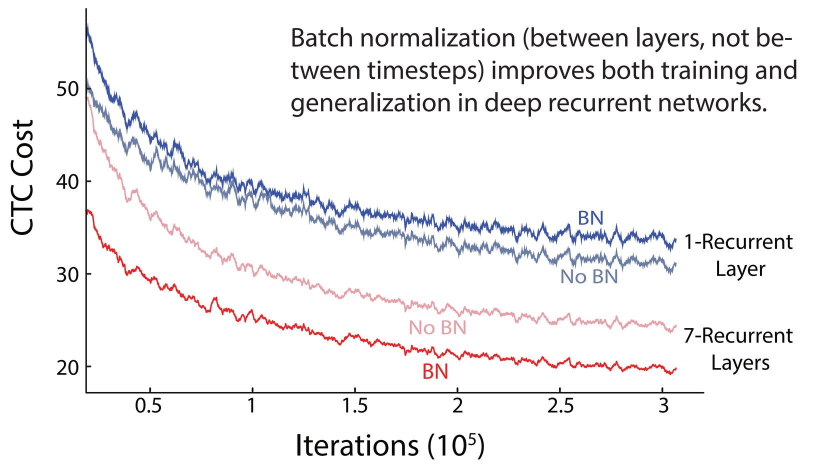
To train on 12,000 hours of speech, we perform many systems optimizations. Placing all computation on distributed GPUs, we can perform synchronous SGD at 3 TFLOP/s per GPU for up to a 128 GPUs (weak scaling).

For deployment, we develop a batching scheduler to improve computational efficiency while minimizing latency. We also create specialized matrix multiplication kernels to perform well at small batch sizes.

Combined with forward only variants of our research models, we achieve a low-latency production system with little loss in recognition accuracy.

Architecture Search / Data Collection

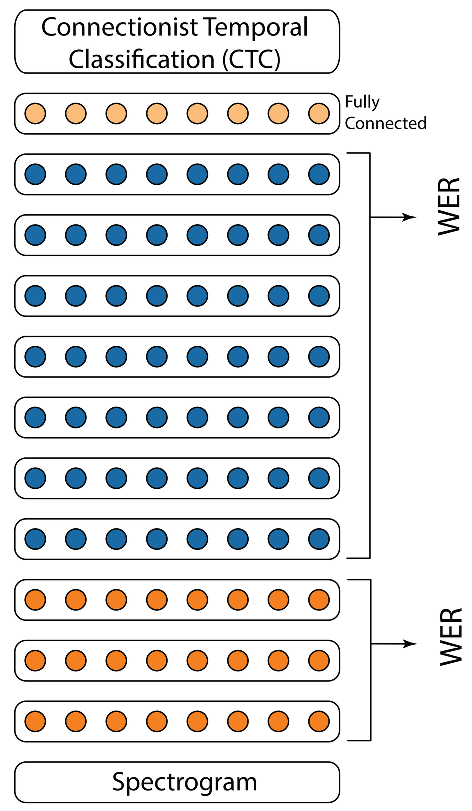
Batch Normalization



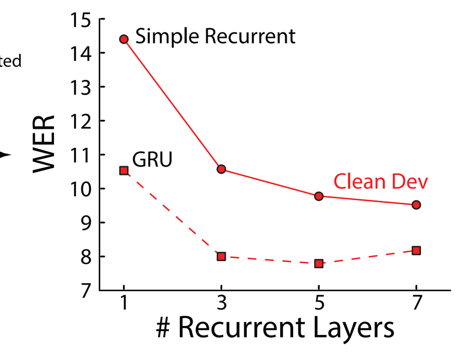
Batch normalization (between layers, not between timesteps) improves both training and generalization in deep recurrent networks.

Layers		Train		Dev	
(Recurrent/Total)	Width	No BN	BN	No BN	BN
1/5	2400	10.55	11.99	13.55	14.40
3/5	1880	9.55	8.29	11.61	10.56
5/7	1510	8.59	7.61	10.77	9.78
7/9	1280	8.76	7.68	10.83	9.52

WER



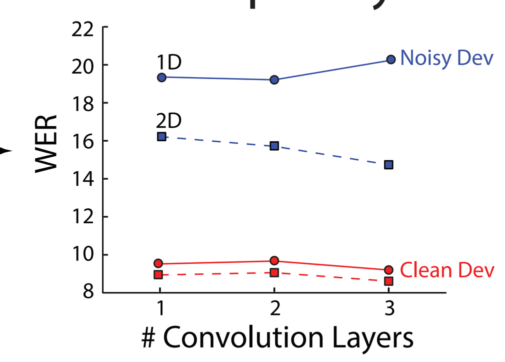
Deep Recurrence



More layers of recurrence helps for both Gated Recurrent Units (GRUs) and Simple Recurrent Units. Increase at 7xGRU is likely due to not enough parameters (layers too thin).

1x1D Convolution Layer (38M Params)

Frequency Convolutions



More layers of 2D convolution generalize better in noisy environments, while more layers of 1D convolution lead to worse generalization.

Temporal Stride: 2
Frequency Stride: 2/layer
7-Recurrent Layers (38M Params)

SortaGrad

	Baseline	Batch Norm
Not Sorted	11.96	9.78
Sorted	10.83	9.52

WER

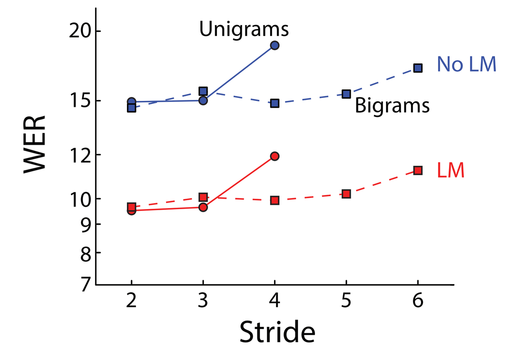
"A Sort of Stochastic Gradient Descent" Curriculum learning with utterance length as a proxy for difficulty. Sorting the first epoch by length improves convergence stability and final values.

Diverse Datasets

Dataset	Speech Type	Hours
WSJ	read	80
Switchboard	conversational	300
Fisher	conversational	2000
LibriSpeech	read	960
Baidu	read	5000
Baidu	mixed	3600
Total		11940

Mix of conversational, read, and spontaneous. Average utterance length ~6 seconds.

Bigrams

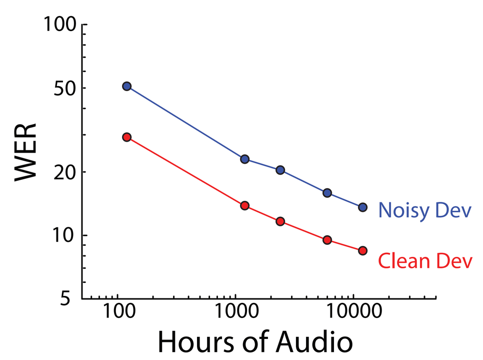


Inspired by Mandarin, we can maintain performance while reducing the number of timesteps (thus dramatically reducing computation) by creating a larger alphabet (bigrams) with fewer characters / second. This decreases the entropy density of the language, allowing for more temporal compression. (ex. [the_cats_sat] -> [th, e, _, ca, ts, _, sa, t])

1x1D Convolution
7-Recurrent Layers (38M Params)

	Unigram	Mandarin	Bigram
Chars / Sec	14.1	3.2	2.4
Bits / Char	4.1	12.6	9.6
Bits / Sec	58.1	40.7	23.3

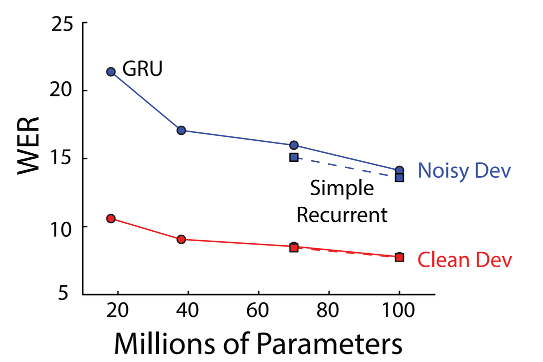
Data Scaling



WER decreases following an approximate power law with amount of training audio (~40% relative / decade). There is a near constant relative offset between noisy and clean generalization (~60%).

2x2D Convolution
Temporal Stride: 2
Frequency Stride: 2/layer
7-Recurrent Layers (70M Params)

Model Size



GRU:
1x2D Convolution
Temporal Stride: 3
Frequency Stride: 2
3-Recurrent Layers
Bigrams

Simple Recurrent:
3x2D Convolution
Temporal Stride: 3
Frequency Stride: 2/layer
7-Recurrent Layers
Bigrams

Single Model Approaching Human Performance on Many Test Sets

Read Speech			
Test set	DS1	DS2	Human
WSJ eval'92	4.94	3.60	5.03
WSJ eval'93	6.94	4.98	8.08
LibriSpeech test-clean	7.89	5.33	5.83
LibriSpeech test-other	21.74	13.25	12.69

WER

Accented Speech			
Test set	DS1	DS2	Human
VoxForge American-Canadian	15.01	7.55	4.85
VoxForge Commonwealth	28.46	13.56	8.15
VoxForge European	31.20	17.55	12.76
VoxForge Indian	45.35	22.44	22.15

WER

Noisy Speech			
Test set	DS1	DS2	Human
CHiME eval clean	6.30	3.34	3.46
CHiME eval real	67.94	21.79	11.84
CHiME eval sim	80.27	45.05	31.33

WER

DS 1 *Hannun et al., Deep speech: Scaling up end-to-end speech recognition, 2014. http://arxiv.org/abs/1412.5567*
1x1D Convolution
Temporal Stride: 2
Unigrams / No BN
1-Recurrent Layer (38M Params)

DS 2 *This Work*
3x2D Convolution (Bigrams / BN)
Temporal Stride: 3
Frequency Stride: 2/layer
7-Recurrent Layers (100M Params)

Human
Amazon mechanical turk transcription. Each utterance was transcribed by 2 turkers, averaging 30 seconds of work time per an utterance.

Architectural Improvements Generalize to Mandarin ASR

Dataset	Type	Hours
Unaccented Mandarin	spontaneous	5000
Accented Mandarin	spontaneous	3000
Assorted Mandarin	mixed	1400
Total		9400

Mix of conversational, read, and spontaneous. Average utterance length ~3 seconds.

Language	Architecture	Dev No LM	Dev LM
English	5-layers, 1 recurrent	27.79	14.39
English	9-layers, 7 recurrent	14.93	9.52
Mandarin	5-layers, 1 recurrent	9.80	7.13
Mandarin	9-layers, 7 recurrent	7.55	5.81

WER English / CER Mandarin

3x2D Convolution
Temporal Stride: 4
Frequency Stride: 2/layer
70M Params

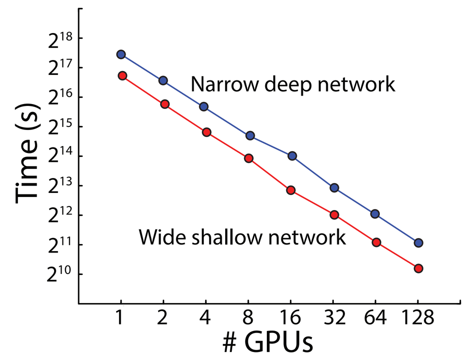
Architecture	Dev	Test
5-layers, 1 recurrent	7.13	15.41
5-layers, 3 recurrent	6.49	11.85
5-layers, 3 recurrent + BN	6.22	9.39
9-layers, 7 recurrent + BN + 2D conv	5.81	7.93

CER

Internal Mandarin voice search test set.

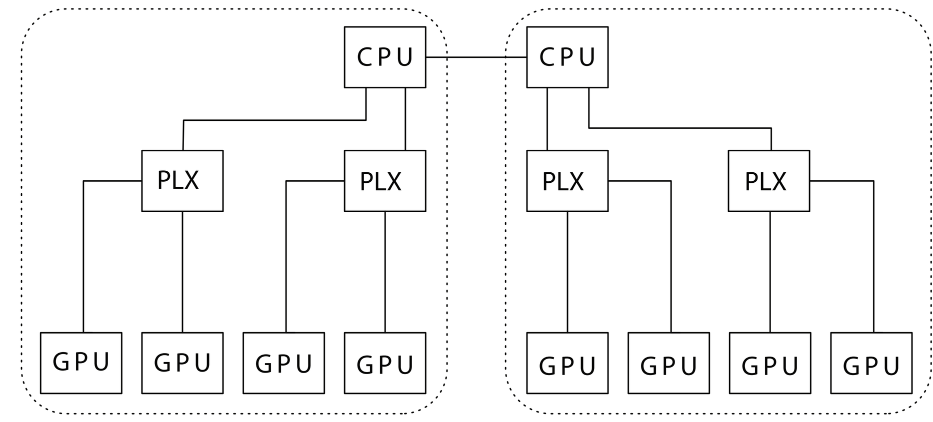
Systems Optimizations / Deployment

Scaling Neural Network Training



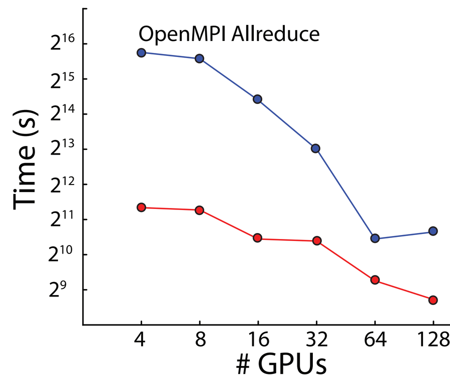
We built a system to train deep recurrent neural networks that can linearly scale from 1 to 128 GPUs, while sustaining 3 TFLOP/s computational throughput per GPU throughout an entire training run (weak scaling). We did this by carefully designing our node, building an effcient communication layer and writing a very fast implementation of CTC cost function so that our entire network runs on the GPU.

Node Architecture



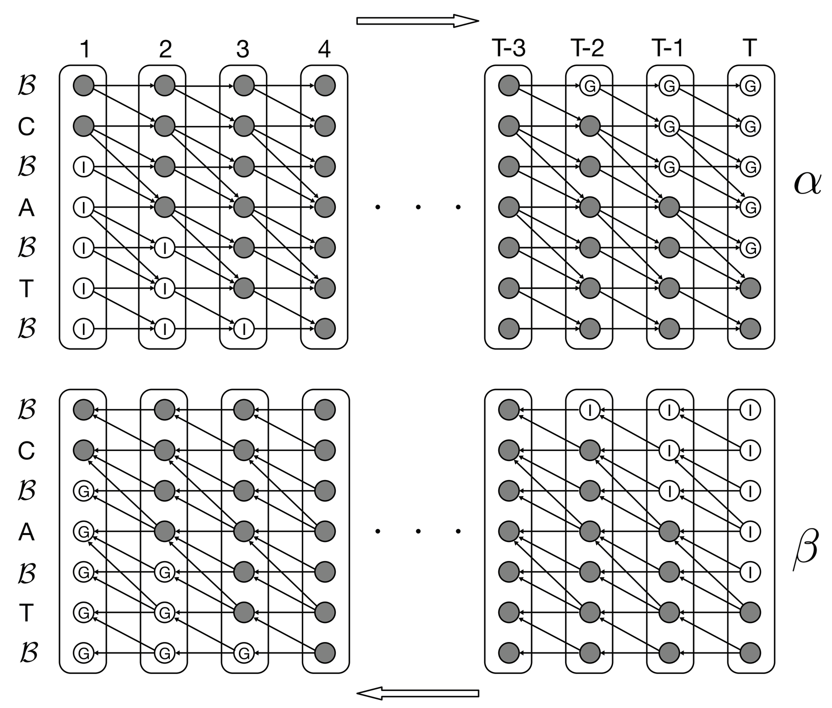
Optimized compute dense node with 53 TFLOP/s peak compute and large PCI root complex for efficient communication

Fast MPI AllReduce



We use synchronous SGD for deterministic results from run to run. The weights are synchronized through MPI Allreduce. We wrote our own Allreduce implementation that is 20x faster than OpenMPI's Allreduce up to 16 GPUs. It can scale up to 128 GPUs and maintain its superior performance

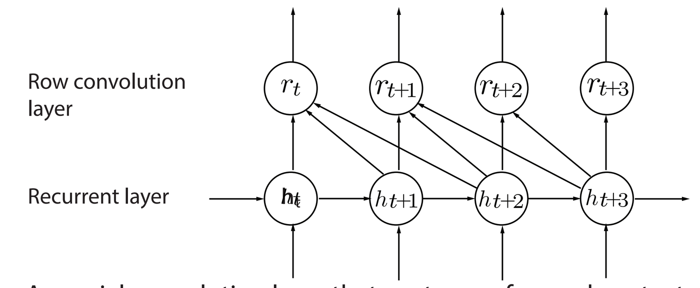
Computing CTC on GPU



Language	CPU CTC Time	GPU CTC Time	Speedup
English	5888.12	203.56	28.9
Mandarin	1688.01	135.05	12.5

An optimized CTC implmentation on the GPU is significantly faster than the corresponding CPU implementation. This also eliminates all roundtrip copies beween CPU and GPU. All times are in seconds for training one epoch of a 5 layer, 3 bi-directional layer network.

Row Convolutions



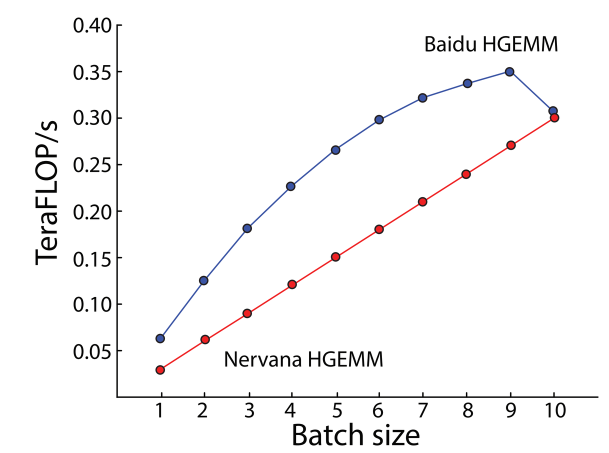
A special convolution layer that captures a forward context of fixed number of timesteps into the future. This allows us to train and deploy a model with only forward recurrent layers. Models with only forward recurrent layers satisfy the latency constraints for deployment since we do not need to wait for the end of utterance.

As we see on the right, the production system that only has forward recurrent layers is very close in performance to the research system with bi-directional layers, on two different test sets.

System	Clean	Noisy
Research	6.05	9.75
Production	6.10	10.00

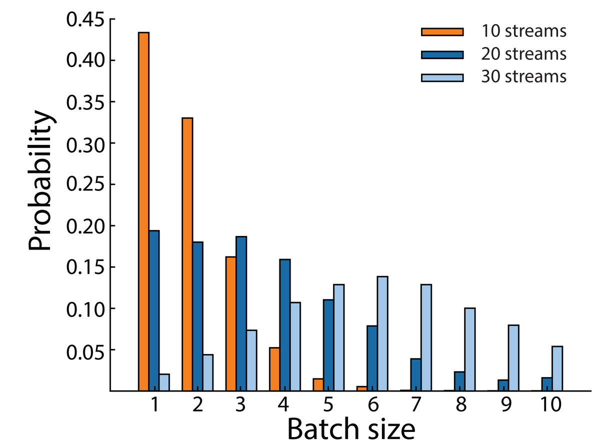
CER

Deployment Optimized Matrix-Matrix Multiply



We wrote a speical matrix-matrix multiply kernel that is efficient for half-precision and small batch sizes. The deployment system uses small batches to minimize latency and uses half-precision to minimize memory footprint. Our kernel is 2x faster than the publicly available kernels from Nervana systems

Batch Dispatch



We built a batching scheduler that assembles streams of data from user requests into batches before doing forward propagation. Large batch size results in more computational efficiency but also higher latency. As expected, batching works best when the server is heavily loaded: as load increases, the distribution shifts to favor processing requests in larger batches. Even for a light load of only 10 concurrent user requests, our system performs more than half the work in batches with at least 2 samples.