Backing off toward simplicity

Understanding the limits of deep learning

Stephen Merity
Salesforce Research
Deep learning is a jackhammer
Deep learning is a jackhammer

- It can be difficult to use (hyper parameters)
  - Miss out on far stronger results due to lack of tuning
Deep learning is a jackhammer

• It can be difficult to use (hyper parameters)
  - Miss out on far stronger results due to lack of tuning

• It can be expensive and slow (GPU++)
  - Poor iteration speed almost invariably leads to worse results
  - In many situations, DL doesn’t lend itself to quick experiments
Deep learning is a jackhammer

• It can be difficult to use (hyper parameters)
  - Miss out on far stronger results due to lack of tuning

• It can be expensive and slow (GPU++)
  - Poor iteration speed almost invariably leads to worse results
  - In many situations, DL doesn’t lend itself to quick experiments

• It’s the wrong or overpowered tool in many situations
  - You *could* make scrambled eggs with a jackhammer but … O_o
Deep learning is a jack hammer

Referring to Pixar’s rendering times, Blinn’s law: “As technology advances, rendering time remains constant”

This is annoying but less of an issue for Pixar artists as:
- Most animators can see their animation in real time
- Lighting artists can render smaller representative images quickly
- …
- **Key point:** individuals can still be productive even when the final render itself might be horribly slow
Deep learning is a jack hammer

Referring to Pixar’s rendering times, Blinn’s law: “As technology advances, rendering time remains constant”

Eric Jang's deep learning (DL) modification to Blinn’s law: "As technology advances, the time to train a deep neural network remains constant."

Unlike Pixar’s artists, deep learning practitioners can’t see a fast approximate result, so it’s a major impact on productivity
- Scaling up or down a model is not predictable or representative
Deep learning take-aways

• Deep learning isn’t always the best tool for the job

• Even when DL is the best tool, you should be strategic:
  - Know what elements give the accuracy gains
  - Know when you’re sacrificing speed / generality / …

• A broad tactic:
  - Start with a fast and well tuned baseline that tackles your task
  - Take deliberate and slow steps towards a SotA model
    (SotA = state of the art)
Deep learning take-aways

• Deep learning isn’t always the best tool for the job

• Even when DL is the best tool, you should be strategic:
  - Know what elements give the accuracy gains
  - Know when you’re sacrificing speed / generality / …

• A broad tactic:
  - Start with a **fast** and **well tuned** baseline that tackles your task
  - Take deliberate and slow steps towards a SotA model
    (SotA = state of the art)

• Stop throwing matrix multiplies at everything without thinking ;)
Overview

• Deep learning is a jackhammer

• **When to use deep learning**

• Most DL models are overpowered

• “Streamlined” DL examples
  - LSTM language modeling
  - Quasi-Recurrent Neural Network
  - Brief note: Google Neural Machine Translation
Deep learning for vision

Deep learning has generally “won” in vision:

• DL has substantial wins over traditional ML techniques

• Parallelization is trivial for most models (= fast training)

• Transfer learning works effectively
  - Train once on a large and general dataset, apply to smaller dataset
Deep learning for sequences

- Text
- Speech
- Time Series
- Logfiles/anomaly detection
- Finance
Deep learning for sequences

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>e.g. 10s of millions of sentence pairs</td>
</tr>
<tr>
<td>Speech</td>
<td>e.g. 10,000 hours of audio</td>
</tr>
<tr>
<td>Time Series</td>
<td></td>
</tr>
<tr>
<td>Logfiles/anomaly detection</td>
<td>e.g. multiple years of server logs</td>
</tr>
<tr>
<td>Finance</td>
<td>e.g. decades of high-resolution prices</td>
</tr>
</tbody>
</table>
Deep learning for sequences

- Text  
  e.g. 10s of millions of sentence pairs

- Speech  
  e.g. 10,000 hours of audio

- Time Series  
  - Logfiles/anomaly detection  
    e.g. multiple years of server logs
  - Finance  
    e.g. decades of high-resolution prices
Deep learning for text

When to use DL for text is far more complicated than for vision:

• Traditional ML techniques can be quite effective in many tasks
  - Pareto principle: 80% of the accuracy in 20% of the time

• DL can be more troublesome when applied to smaller datasets

• Discards the majority of well defined and justified formalisms in NLP

• Transfer learning for text is still naive*
  ([GloVe, word2vec, …] or Contextualized Word Vectors (CoVe))
Bad for DL: sentiment classification

Using an LSTM for long form sentiment classification is a Bad Idea™

• Minimal supervision signal: [strong neg, neg, pos, strong pos]

• Vanishing gradient over many hundreds of time steps
  - The neural network has no signal to words in the distant past

• Difficult to capture the simplest and most important details: n-grams
  - “… hate the film …”
  - “… disappointing rendition of …”
  - “… the wooden acting was …”
Bad for DL: sentiment classification

Using an LSTM for long form sentiment classification is a Bad Idea™

- Difficult to capture the simplest and most important details: n-grams
  - “… hate the film” / “… disappointing rendition …” / “… wooden acting …”

- The film was full of wooden acting and … <word> <word> <word> <word>
  <word> <word> <word> <word> <word> <word> <word> <word> <word> <word> <word>
  <word> <word> <word> <word> <word> <word> <word> <word> <word> <word> <word>
  <word> <word> <word> <word> <word> <word> <word> <word> <word> <word> <word>
  <word> <word> <word> <word> <word> <word> <word> <word> <word> <word> <word>
  = NEGATIVE

- This minimal signal have to go back HUNDREDS of timesteps, through complex hidden state transformations, to update a word vector of hundreds of parameters

- Delip Rao mentioned that “All of DL is learning representations from data” … and this is a worst case scenario for deep learning to learn them..!
Good for ML: sentiment classification

Using n-gram features and a traditional classifier will get you 90+% of the accuracy and train in a few seconds …

ngram(“I hate the film. A disappointing rendition with wooden acting.”) => [“I”, “hate”, “the”, … “disappointing rendition”, “rendition with”, “with wooden”, “wooden acting”]

Throwing this into your friendly traditional ML classifier (logistic regression, SVM, Naive Bayes, …) will get you fast and reasonable results
ML vs DL: sentiment classification

- You could make your architecture a mix of the two (Google refer to it as “wide and deep”)
  - Wide = simple classifier with n-gram or categorical features
  - Deep = LSTM with full word vectors
  … but you better know why - as otherwise all you get is a slow model that gets a slightly better result than normal!
ML vs DL: sentiment classification

A simple shallow model, Naive Bayes SVM (NBSVM), from “Baselines and Bigrams” (Wang and Manning, 2012)

• Trains in seconds on a CPU  
  (in comparison to the minutes / hours an LSTM on GPU needs)

• See Jeremy Howard’s NBSVM++ PyTorch code:  
  - Fits in a few lines, simple, and fast  
  - Strong competitive results on IMDB sentiment task
Overview

• Deep learning is a jackhammer

• When to use deep learning

• **Most DL models are overpowered**

• “Streamlined” DL examples
  - LSTM language modeling
  - Quasi-Recurrent Neural Network
  - Brief note: Google Neural Machine Translation
Most DL models are overpowered

Controversial claim: Most deep learning models are *overpowered* for what they need
Most DL models are overpowered

Controversial claim:
Most deep learning models are overpowered for what they need

- Many of the results can be achieved with simpler architectures
- With too many moving parts, it's hard to isolate improvements
- The models are slower to train than necessary
- These results can mislead human intuition, which is already fragile

Better to use compute intelligently than throw more of it at the problem!
Be strategic with deep learning

• Ensure you make the model no more complex than you need it to be  
  - If in doubt, start overly simple (you might surprise yourself!)

• Push your **simple** and **fast** baseline model as far as possible  
  (tune hyperparams, regularization techniques, data processing, …)

• Inch step by step toward the more complex models:  
  add components one at a time, evaluate, and continue
Baselines deserve more love

- Baselines help both you and others understand the model and data
  - Data processing issues are easily hidden by complex models

- Sanity check: your visual question answering (VQA) results should get better when you add the image
  “However, inherent structure in our world and bias in our language tend to be a simpler signal for learning than visual modalities, resulting in models that ignore visual information, leading to an inflated sense of their capability.” - Goyal et al. 2016

- Key point:
  don’t trust yourself, your models, or your intuition :)


Human intuition on DL is broken

• Larger hidden states were originally considered more important than the use of attention (and was only reversed when results got better)

• PixelCNNs / quasi-recurrent neural networks (QRNNs) / ByteNet / etc all show that full recurrence in recurrent neural networks isn’t necessary

• “Attention is all you need” questions whether recurrence is necessary at all

• Remember: neural networks were considered “silly” in the past too (“Neural networks only find local optima and are unstable and difficult to train”)
Overview

• Deep learning is a jackhammer

• When to use deep learning

• Most DL models are overpowered

• “Streamlined” DL examples
  - LSTM language modeling
  - Quasi-Recurrent Neural Network
  - Brief note: Google Neural Machine Translation
Language modeling

- For most of 2016, the state of the art in language modeling involved custom RNN architectures far more complicated than the LSTM (*lower is better*)

<table>
<thead>
<tr>
<th>Model Description</th>
<th>Parameters</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>24M</td>
<td>75.7</td>
<td>73.2</td>
</tr>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>51M</td>
<td>71.1</td>
<td>68.5</td>
</tr>
<tr>
<td>Zilly et al. (2016) - Variational RHN (tied)</td>
<td>23M</td>
<td>67.9</td>
<td>65.4</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td>25M</td>
<td>-</td>
<td>64.0</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td>54M</td>
<td>-</td>
<td>62.4</td>
</tr>
</tbody>
</table>

- Recurrent Highway Network (RHN) is similar but slower than the LSTM: updates the hidden state many times per timestep

- Zoph et al’s RNN architecture was made using tens of thousands of experiments
Language modeling

• Recurrent Highway Network (RHN), although no longer state of the art, had released an open source implementation (huzzah!)

• Zoph et al’s RNN architecture (NASCell) was mysterious:
  - the cell architecture was quite complex with no human intuition (generated by a reinforcement learning algorithm)
  - didn’t list hyper parameters used in experiments (still not available)
  - had no open source implementation of the full experiment
Language modeling

• I decided to improve PyTorch’s baseline LM example, hoping to get near the current state of the art results for the LSTM

• Goal was to modify the model minimally and maintain speed
  - Ensure the model remained simple for teaching purposes
  - Any changes must keep using NVIDIA’s cuDNN LSTM (= fast)

• Couldn’t change the RNN architecture as that would be far slower: cuDNN LSTM is highly optimized (6.5-11x faster) but not flexible
LM: standard regularization

• Apply L2 weight decay

• Apply L2 regularization to the model’s activations:
  - The RNN’s activation (i.e. the output of the RNN)
  - The difference between RNN activation at timestep $t$ and $t+1$

<table>
<thead>
<tr>
<th>Model</th>
<th>24M</th>
<th>51M</th>
<th>23M</th>
<th>25M</th>
<th>54M</th>
<th>PTB, LSTM (tied) $h=650$, $\alpha=5$, $\beta=2$, $dp=0.5$, $dp_h=0.4$</th>
<th>13M</th>
<th>24M</th>
<th>51M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>75.7</td>
<td>71.1</td>
<td>67.9</td>
<td>-</td>
<td>67.9</td>
<td>72.0</td>
<td>68.9</td>
<td>70.2</td>
<td>68.2</td>
</tr>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>73.2</td>
<td>68.5</td>
<td>65.4</td>
<td>-</td>
<td>62.4</td>
<td>68.9</td>
<td>66.9</td>
<td>65.4</td>
<td>65.4</td>
</tr>
<tr>
<td>Zilly et al. (2016) - Variational RHN (tied)</td>
<td></td>
<td></td>
<td>65.4</td>
<td>-</td>
<td>62.4</td>
<td>68.2</td>
<td>68.9</td>
<td>66.9</td>
<td>65.4</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

… Wait a second … Really ..? Hmm … :)
LM: recurrent regularization

• Next needed step: recurrent regularization
  - Essentially necessary for SotA language modeling on smaller dataset

• Why is recurrent regularization required?
  Standard dropout doesn’t work on an RNN’s hidden state

(a) Naive dropout RNN
LM: recurrent regularization

• Next needed step: *recurrent regularization*
  - Essentially necessary for SotA language modeling on smaller dataset

• Why is recurrent regularization required?
  Standard dropout doesn’t work on an RNN’s hidden state

(a) Naive dropout RNN
Variational dropout

- Variational dropout (Gal and Ghahramani, 2015) “locks” the dropout mask
- Prevents excessive loss on the hidden state, works incredibly well, is only a dozen lines of code - assuming you can modify your LSTM

(a) Naive dropout RNN
(b) Variational RNN
LM: recurrent dropout

- **Issue:** modifying the LSTM isn’t an option as:
  - The NVIDIA cuDNN LSTM isn’t flexible
  - Losing the NVIDIA cuDNN LSTM results in a 6-11x slower model :’(

- **Key question:** how do we get the gains without sacrificing the speed?
Weight dropped LSTM

• In the NVIDIA cuDNN LSTM, we can’t change any of the code … but we can fiddle with the LSTM weights as much as we’d like!

• By applying dropout to the LSTM weights, similar to DropConnect, we can achieve a similar result to variational dropout (indeed, we can achieve exactly variational dropout, but found weight dropped LSTM achieved better results)
Final language modeling results

- These techniques, plus some others (see our paper for full details), gave us state of the art results with the same number of parameters.

- Independently of our work, Melis et al. 2017 found similar results using a mostly unmodified LSTM.

<table>
<thead>
<tr>
<th>Model Description</th>
<th>Parameters</th>
<th>BLEU</th>
<th>ROUGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>24M</td>
<td>75.7</td>
<td>73.2</td>
</tr>
<tr>
<td>Inan et al. (2016) - Variational LSTM (tied) + augmented loss</td>
<td>51M</td>
<td>71.1</td>
<td>68.5</td>
</tr>
<tr>
<td>Zilly et al. (2016) - Variational RHN (tied)</td>
<td>23M</td>
<td>67.9</td>
<td>65.4</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td>25M</td>
<td>—</td>
<td>64.0</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016) - NAS Cell (tied)</td>
<td>54M</td>
<td>—</td>
<td>62.4</td>
</tr>
<tr>
<td>Melis et al. (2017) - 4-layer skip connection LSTM (tied)</td>
<td>24M</td>
<td>60.9</td>
<td>58.3</td>
</tr>
<tr>
<td>AWD-LSTM - 3-layer LSTM (tied)</td>
<td>24M</td>
<td>60.0</td>
<td>57.3</td>
</tr>
</tbody>
</table>
Final language modeling results

- A well tuned baseline can sometimes really surprise you

- My initial reason (improve PyTorch example) meant that I modified the code minimally, tuned the model well, and kept it fast for experimentation

- Paper: Regularizing and Optimizing LSTM Language Models
  Code for PyTorch: [https://github.com/salesforce/awd-lstm-lm](https://github.com/salesforce/awd-lstm-lm)

<table>
<thead>
<tr>
<th></th>
<th>Model Configuration</th>
<th>Param (M)</th>
<th>Perplexity (log)</th>
<th>BLEU (3-gram)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inan et al. (2016)</td>
<td>Variational LSTM (tied) + augmented loss</td>
<td>24M</td>
<td>75.7</td>
<td>73.2</td>
</tr>
<tr>
<td>Inan et al. (2016)</td>
<td>Variational LSTM (tied) + augmented loss</td>
<td>51M</td>
<td>71.1</td>
<td>68.5</td>
</tr>
<tr>
<td>Zilly et al. (2016)</td>
<td>Variational RHN (tied)</td>
<td>23M</td>
<td>67.9</td>
<td>65.4</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016)</td>
<td>NAS Cell (tied)</td>
<td>25M</td>
<td>—</td>
<td>64.0</td>
</tr>
<tr>
<td>Zoph &amp; Le (2016)</td>
<td>NAS Cell (tied)</td>
<td>54M</td>
<td>—</td>
<td>62.4</td>
</tr>
<tr>
<td>Melis et al. (2017)</td>
<td>4-layer skip connection LSTM (tied)</td>
<td>24M</td>
<td>60.9</td>
<td>58.3</td>
</tr>
<tr>
<td>AWD-LSTM - 3-layer LSTM (tied)</td>
<td></td>
<td>24M</td>
<td>60.0</td>
<td>57.3</td>
</tr>
</tbody>
</table>
Overview

• Deep learning is a jackhammer

• When to use deep learning

• Most DL models are overpowered

• “Streamlined” DL examples
  - LSTM language modeling
  - Quasi-Recurrent Neural Network
  - Brief note: Google Neural Machine Translation
Case study: speeding up LSTMs

• Recurrent neural networks, specifically LSTMs, are used extensively in DL

• Problems:
  - RNNs make it hard to fully utilize the parallelism of the GPU
  - NVIDIA cuDNN RNNs are not flexible (i.e. recurrent regularization, …)

Used with permission from Chris Olah’s blog
Case study: speeding up LSTMs

- Question: what is responsible for the slowness of an LSTM and is that bottlenecked component actually necessary?
  Answer: the slow sequential nature (see red boxes in LSTM/Linear)
LSTM in detail

Issue: the highlighted blocks use the previous hidden state and force the model to be sequential

\[ z_t = \tanh(W_z x_t + V_z h_{t-1} + b_z) \]
\[ i_t = \text{sigmoid}(W_i x_t + V_i h_{t-1} + b_i) \]
\[ f_t = \text{sigmoid}(W_f x_t + V_f h_{t-1} + b_f) \]
\[ o_t = \text{sigmoid}(W_o x_t + V_o h_{t-1} + b_o) \]
\[ c_t = i_t \odot z_t + f_t \odot c_{t-1} \]
\[ h_t = o_t \odot \tanh(c_t) \]
LSTM in detail

\[ z_t = \tanh(W_z x_t + V_z h_{t-1} + b_z) \]
\[ i_t = \text{sigmoid}(W_i x_t + V_i h_{t-1} + b_i) \]
\[ f_t = \text{sigmoid}(W_f x_t + V_f h_{t-1} + b_f) \]
\[ o_t = \text{sigmoid}(W_o x_t + V_o h_{t-1} + b_o) \]
\[ c_t = i_t \odot z_t + f_t \odot c_{t-1} \]
\[ h_t = o_t \odot \tanh(c_t) \]
Optimizing the LSTM

• Individual components of LSTM equations are fast on GPU
  - Matrix multiply, vector add, tanh, sigmoid, etc.

• But the overall cell computation isn’t a natural GPU workload
  - Many small matrix multiply (BLAS) calls
  - Throughput limited by CUDA kernel / cuBLAS launch overhead
Back to the fundamental problem

- It’s hard to effectively utilize the GPU with lots of small BLAS calls
- We want an architecture that’s inherently more parallel …
How do ConvNets solve this?

- In the world of images and video, there’s a fully parallel approach

Used with permission from Chris Olah’s blog
How do ConvNets solve this?

- In the world of images and video, there’s a fully parallel approach.
- But sequence data is usually much more sensitive to ordering!

Used with permission from Chris Olah’s blog.
Solution: quasi-recurrent architectures

Use a convolution but replace the pooling component with something that’s
(a) fast and (b) sensitive to order
QRNN in detail

- Start with 1D convolution
- parallel across timesteps
- produces all values, including gates + candidate updates

All that needs to be computed recurrently is a simple element-wise pooling function inspired by the LSTM

- Can be fused across time without having to alternate with BLAS operations

\[ z_t = \tanh(W_z \ast X + b_z) \]
\[ [i_t = \text{sigmoid}(W_i \ast X + b_i)] \]
\[ f_t = \text{sigmoid}(W_f \ast X + b_f) \]
\[ [o_t = \text{sigmoid}(W_o \ast X + b_o)] \]

\[ f\text{-pooling:} \]
\[ h_t = (1 - f_t) \odot z_t + f_t \odot h_{t-1} \]

\[ f_o\text{-pooling:} \]
\[ c_t = (1 - f_t) \odot z_t + f_t \odot c_{t-1} \]
\[ h_t = o_t \odot \tanh(c_t) \]

\[ i\text{-pooling:} \]
\[ c_t = i_t \odot z_t + f_t \odot c_{t-1} \]
\[ h_t = o_t \odot \tanh(c_t) \]
QRNN in detail

• Start with 1D convolution
  • parallel across timesteps
  • produces all values, including gates + candidate updates
• All that needs to be computed recurrently is a simple element-wise pooling function inspired by the LSTM
• With minimal custom CUDA kernel, can run over hundreds of timesteps in milliseconds

\[
\begin{align*}
z_t &= \tanh(W_z * X + b_z) \\
[i_t &= \text{sigmoid}(W_i * X + b_i)] \\
f_t &= \text{sigmoid}(W_f * X + b_f) \\
[o_t &= \text{sigmoid}(W_o * X + b_o)] \\
\end{align*}
\]

\[
\begin{align*}
f\text{-pooling:} \\
h_t &= (1 - f_t) \odot z_t + f_t \odot h_{t-1} \\
fo\text{-pooling:} \\
c_t &= (1 - f_t) \odot z_t + f_t \odot c_{t-1} \\
h_t &= o_t \odot \tanh(c_t) \\
ifo\text{-pooling:} \\
c_t &= i_t \odot z_t + f_t \odot c_{t-1} \\
h_t &= o_t \odot \tanh(c_t)
\end{align*}
\]
QRNN implementation

- Efficient 1D convolution is built into most deep learning frameworks
- Automatically parallel across time
- Pooling component is implemented in 40 total lines of CUDA C
- Fused across time into one GPU kernel with a simple for loop
## QRNN speed

<table>
<thead>
<tr>
<th>Batch size</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><strong>5.5x</strong></td>
<td><strong>8.8x</strong></td>
<td><strong>11.0x</strong></td>
<td><strong>12.4x</strong></td>
<td><strong>16.9x</strong></td>
</tr>
<tr>
<td>16</td>
<td><strong>5.5x</strong></td>
<td><strong>6.7x</strong></td>
<td><strong>7.8x</strong></td>
<td><strong>8.3x</strong></td>
<td><strong>10.8x</strong></td>
</tr>
<tr>
<td>32</td>
<td><strong>4.2x</strong></td>
<td><strong>4.5x</strong></td>
<td><strong>4.9x</strong></td>
<td><strong>4.9x</strong></td>
<td><strong>6.4x</strong></td>
</tr>
<tr>
<td>64</td>
<td><strong>3.0x</strong></td>
<td><strong>3.0x</strong></td>
<td><strong>3.0x</strong></td>
<td><strong>3.0x</strong></td>
<td><strong>3.7x</strong></td>
</tr>
<tr>
<td>128</td>
<td><strong>2.1x</strong></td>
<td><strong>1.9x</strong></td>
<td><strong>2.0x</strong></td>
<td><strong>2.0x</strong></td>
<td><strong>2.4x</strong></td>
</tr>
<tr>
<td>256</td>
<td><strong>1.4x</strong></td>
<td><strong>1.4x</strong></td>
<td><strong>1.3x</strong></td>
<td><strong>1.3x</strong></td>
<td><strong>1.3x</strong></td>
</tr>
</tbody>
</table>

(slow advantage over NVIDIA cuDNN 5.1 LSTM)
QRNN speed

![QRNN speed diagram](image)
QRNN results

• This simplification, motivated for speed, surely results in worse accuracy, right?

• Apparently not!

• Some ideas as to why:
  • No recurrent weight matrix means no vanishing or exploding gradient problem
  • Elimination of recurrent degrees of freedom is a form of regularization

<table>
<thead>
<tr>
<th>Model</th>
<th>Dev Perplexity</th>
<th>Test Perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>85.7</td>
<td>82.0</td>
</tr>
<tr>
<td>QRNN</td>
<td>82.9</td>
<td>79.9</td>
</tr>
<tr>
<td>QRNN w/zoneout</td>
<td>82.1</td>
<td>78.3</td>
</tr>
</tbody>
</table>

Perplexity on Penn Treebank language modeling dataset (lower is better). All models have 2 layers of 640 hidden units.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time/Epoch (s)</th>
<th>Test Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>480</td>
<td>90.9</td>
</tr>
<tr>
<td>QRNN</td>
<td>150</td>
<td>91.4</td>
</tr>
</tbody>
</table>

Accuracy on IMDb binary sentiment classification dataset. All models have 4 densely-connected layers of 256 hidden units.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time/Epoch (hr)</th>
<th>BLEU (TED.tst2014)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>4.2</td>
<td>16.5</td>
</tr>
<tr>
<td>QRNN</td>
<td>1.0</td>
<td>19.4</td>
</tr>
</tbody>
</table>

Performance on IWSLT German-English character-level machine translation. All models have 4 layers of 320 hidden units; the QRNN’s 1st layer has filter size 6.
Trade-offs?

• You need one or more QRNN layers

• Probably not a good choice for tasks requiring complex hidden state interaction

• Controlling a reinforcement learning agent
QRNN applications

• Baidu’s Silicon Valley AI Lab found QRNNs to give a substantial improvement in accuracy and regularization for a component of their speech generation pipeline (Deep Voice)

• NVIDIA engineers have applied QRNNs to logfile anomaly detection

• A high-frequency trading firm we probably can’t name uses QRNNs

• The language modeling example noted above: similar results while being over two times faster than the tuned NVIDIA cuDNN LSTM
History of limited recurrence architectures

- Similar ideas have been floating around for a long time
- Echo State Networks (Jaeger 2001 through Jaeger 2007)
  - Fix RNN recurrent weights randomly, train only non-recurrent weights
- Our QRNN work was inspired by several papers whose models can be seen as special cases of QRNNs:
  - PixelCNNs (van den Oord et al., January 2016)
  - Strongly Typed RNNs (Balduzzi and Ghifary, February 2016)
  - Query-Reduction Networks (Seo et al., June 2016)
QRNNs

- **Key question:** why are LSTMs slow and can we get rid of that bottleneck?
- Drop-in replacement for LSTM or GRU as a deep learning model for sequences with similar or better accuracy
- Fast even compared to highly tuned custom LSTM kernels
- As an author of the QRNN, their accuracy vs LSTMs surprised even me
  - Reminder: “human intuition” is just not that good!
Overview

- Deep learning is a jackhammer
- When to use deep learning
- Most DL models are overpowered
- "Streamlined" DL examples
  - LSTM language modeling
  - Quasi-Recurrent Neural Network
  - Brief note: Google Neural Machine Translation
Google Neural Machine Translation (GNMT)

- Very point summary of a blog post at Smerity.com: “Peeking into the neural network architecture used for Google's Neural Machine Translation”

- No depth (see the blog post) but key insights …
Why was GNMT important?

- Google sacrificed many of the components in state-of-the-art (SotA) MT architectures in order to allow for faster training
  - They have a single bi-directional LSTM rather than sixteen

- In case you weren’t aware, Google have approximately a bajillion GPUs ;)

- The GNMT framework has since been used in a wide variety of work:
  - Zero shot translation with GNMT
  - Conversational dialogue systems
  - Used in variety of production systems …
Deep learning take-aways

- Deep learning isn’t always the best tool for the job

- Even when DL is the best tool, you should be strategic:
  - Know what elements give the accuracy gains
  - Know when you’re sacrificing speed / generality / …

- A broad tactic:
  - Start with a **fast** and **well tuned** baseline
  - Take deliberate and slow steps towards a SotA model
    (SotA = state of the art)

- Stop throwing matrix multiplies at everything without thinking ;)
Interested in learning more?

Read Salesforce Research’s work online at einstein.ai

Contact me at: @smerity / smerity.com