State-of-the-Art Large Scale Language Modeling in 12 Hours With a Single GPU

Nitish Shirish Keskar - @strongduality
Stephen Merity - @smerity

About Us

Stephen Merity
smerity.com

M.S. from Harvard University (2014)

Research Interests in deep learning:
- Sequence modeling
- Design of efficient architectures
- Memory and pointer networks

Nitish Shirish Keskar
keskarnitish.github.io

PhD from Northwestern University (2017)

Research Interests in deep learning:
- Optimization, generalization and landscapes
- Architecture design and regularization
- Applications in natural language processing

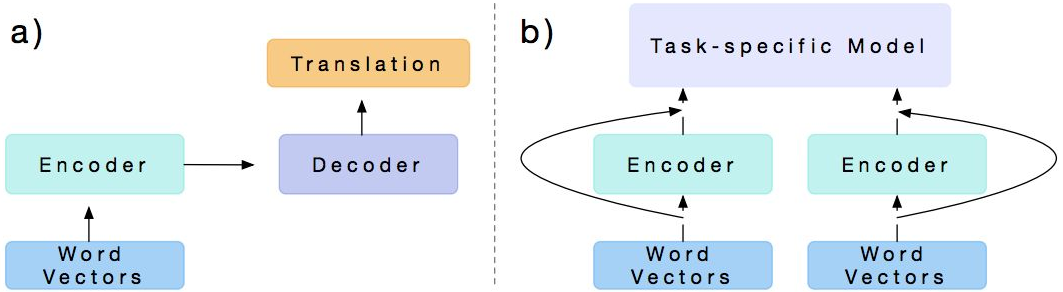
Language modeling

By accurately assigning probability to a natural sequence (words or characters), you can improve:

- **Machine Translation:** p(strong tea) > p(powerful tea)
- **Speech Recognition:** p(speech recognition) > p(speech wreck ignition)
- **Question Answering / Summarization:**
  p(President X attended ...) is higher for X=Obama
- **Query Completion:** p(Michael Jordan Berkeley) > p(Michael Jordan sports)

We can do *far* more than that now however!

Language modeling for transfer learning (beyond embeddings)



Fine-tuned Language Models for Text Classification

Jeremy Howard
fast.ai
University of San Francisco
Singularity University
j@fast.ai

Sebastian Ruder
Insight Centre, NUI Galway
Aylien Ltd., Dublin
sebastian@ruder.io

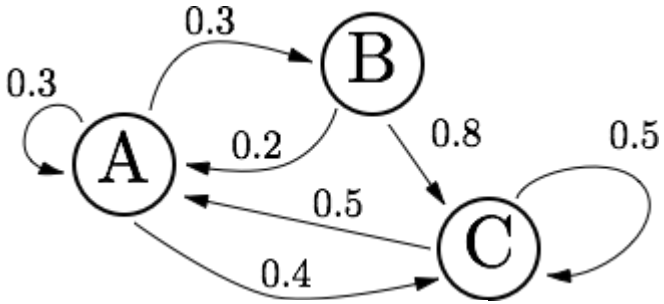
Deep contextualized word representations

Matthew E. Peters[†], **Mark Neumann**[†], **Mohit Iyyer**[†], **Matt Gardner**[†],
{matthewp,markn,mohiti,mattg}@allenai.org

Christopher Clark[*], **Kenton Lee**[*], **Luke Zettlemoyer**[†*]
{csquared,kentonl,lsz}@cs.washington.edu

[†]Allen Institute for Artificial Intelligence
[*]Paul G. Allen School of Computer Science & Engineering, University of Washington

Traditional approaches



n-gram models and their adaptations:

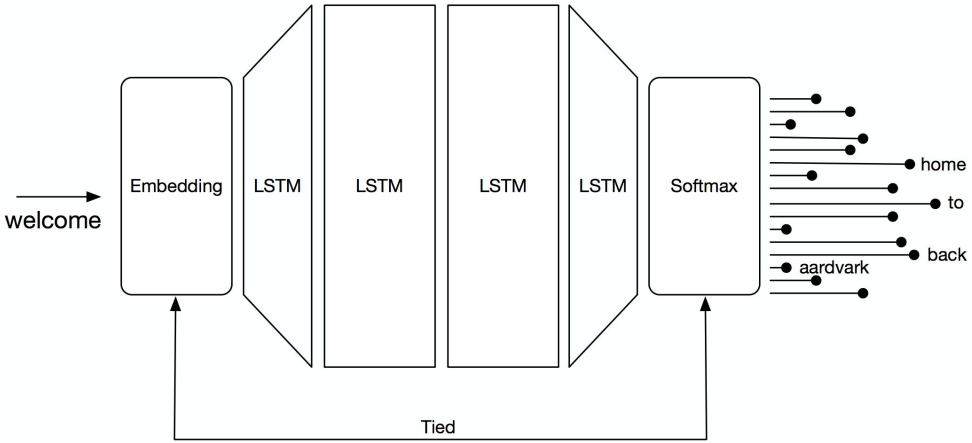
$$
\begin{aligned}
& P(\text{I, saw, the, red, house}) \\
\approx\ & P(\text{I} \mid \langle s \rangle, \langle s \rangle) P(\text{saw} \mid \langle s \rangle, I) P(\text{the} \mid \text{I, saw}) P(\text{red} \mid \text{saw, the}) P(\text{house} \mid \text{the, red}) P(\langle /s \rangle \mid \text{red, house})
\end{aligned}
$$

Issues:

- What to do if n-gram has never been seen?
- How do you choose n for the n-grams?
- “Deep Learning is **amazing** because” v/s “Deep Learning is **awesome** because”

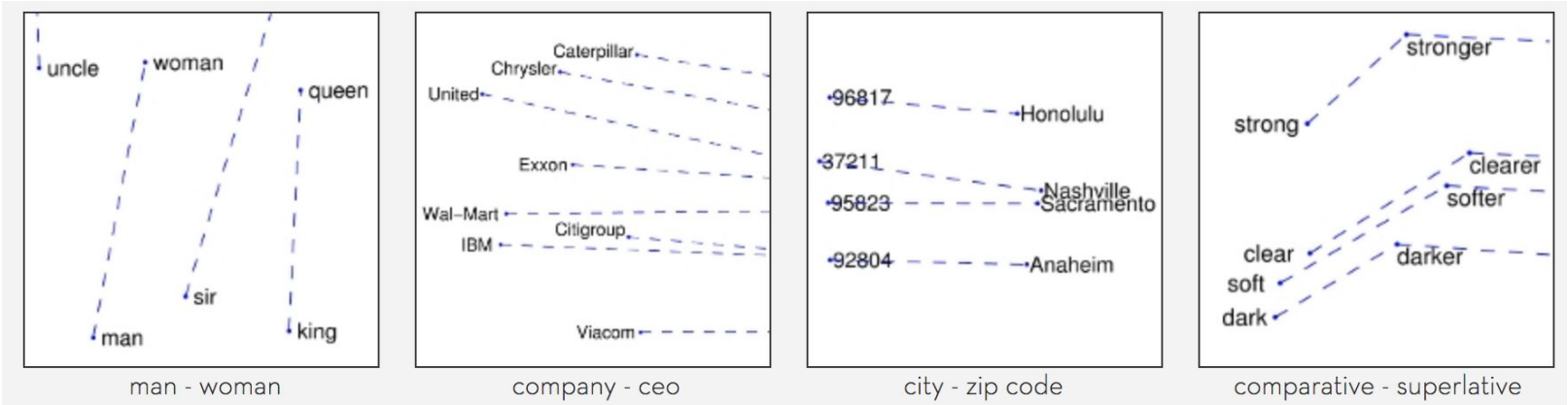
A basic neural language model architecture

- Embedding layer (later - why we need embedding dropout)
- RNN (LSTM, later - QRNN also fits)
- Softmax (“attention” over words,
  later - adaptive softmax for large vocab and efficient GPU)

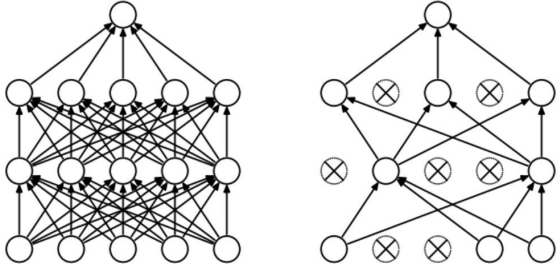


Embeddings

- tl;dr - a vector representation of words
- Every word gets a trainable vector; surprisingly, embeddings create a “geometry” for words.
- First step of neural language modeling; can’t use deep learning without numbers
- Typically, 100-300 dimensional



Embedding Dropout


(a) Standard Neural Net

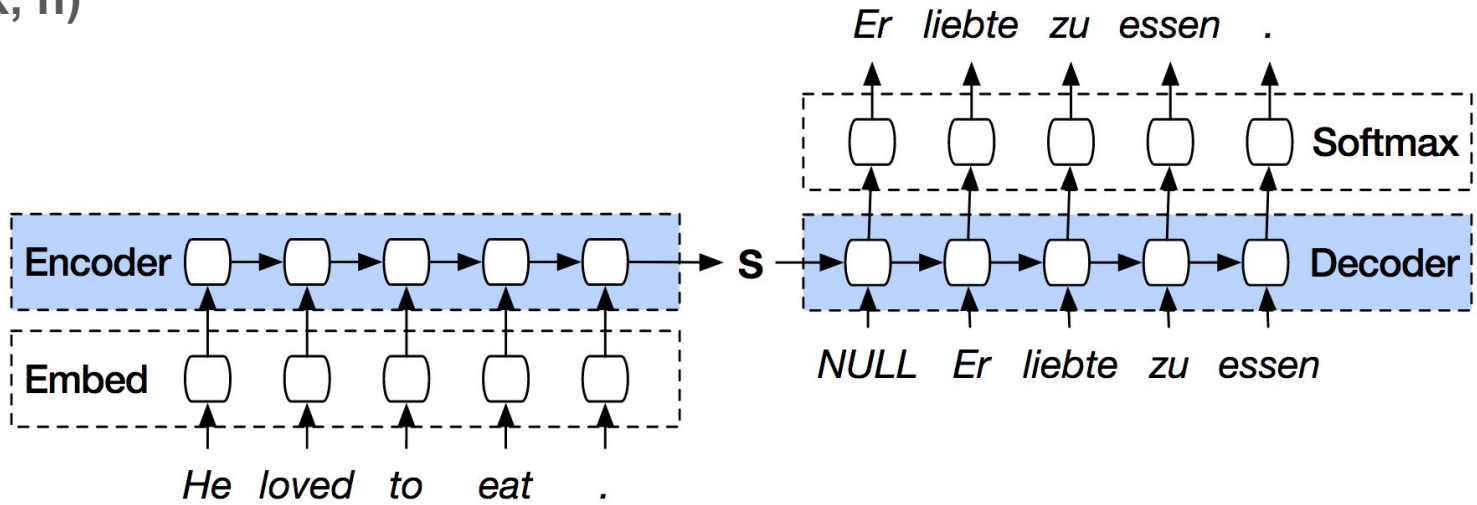
(b) After applying dropout.

- Randomly drop out entire words in the vocabulary!
- Prevents over-fitting in the embeddings

Recurrent Neural Networks

An RNN updates an internal state **h** according to the:
existing state h, the **current input x**, a function **f**

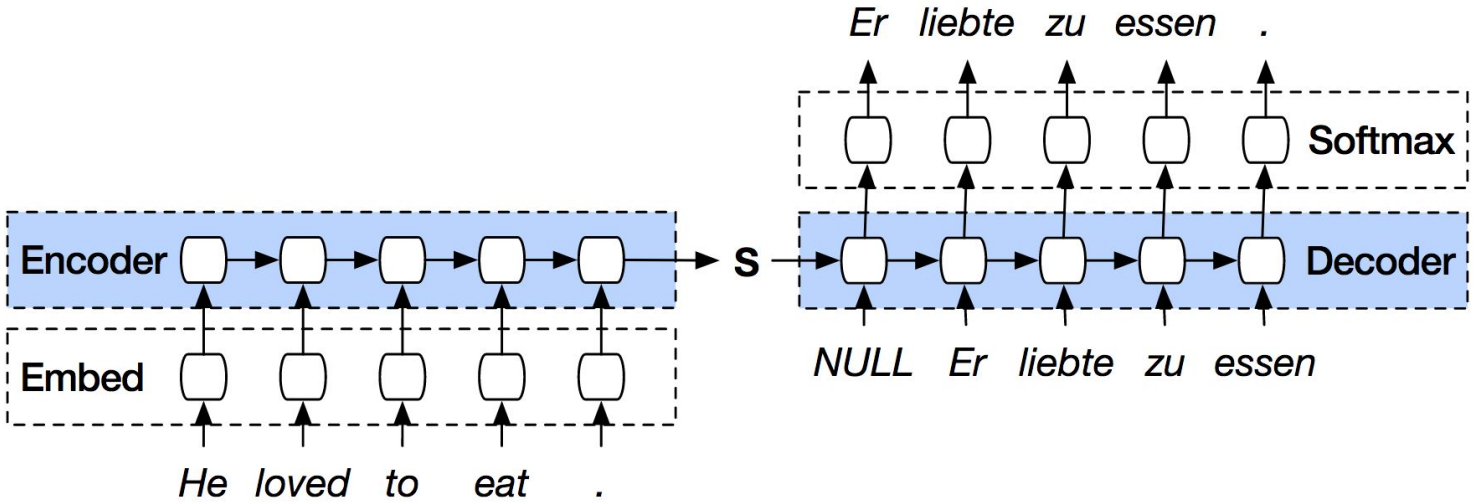
h = f(x, h)



Recurrent Neural Networks

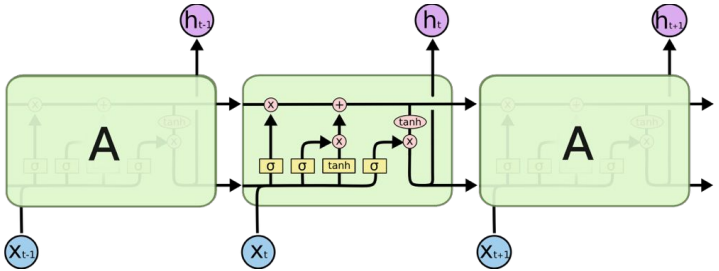
The function **f** can be broken into two parts:

- The transformation of the input **x** to update the hidden state **h**
- The recurrence that updates the new hidden state based on the old hidden state



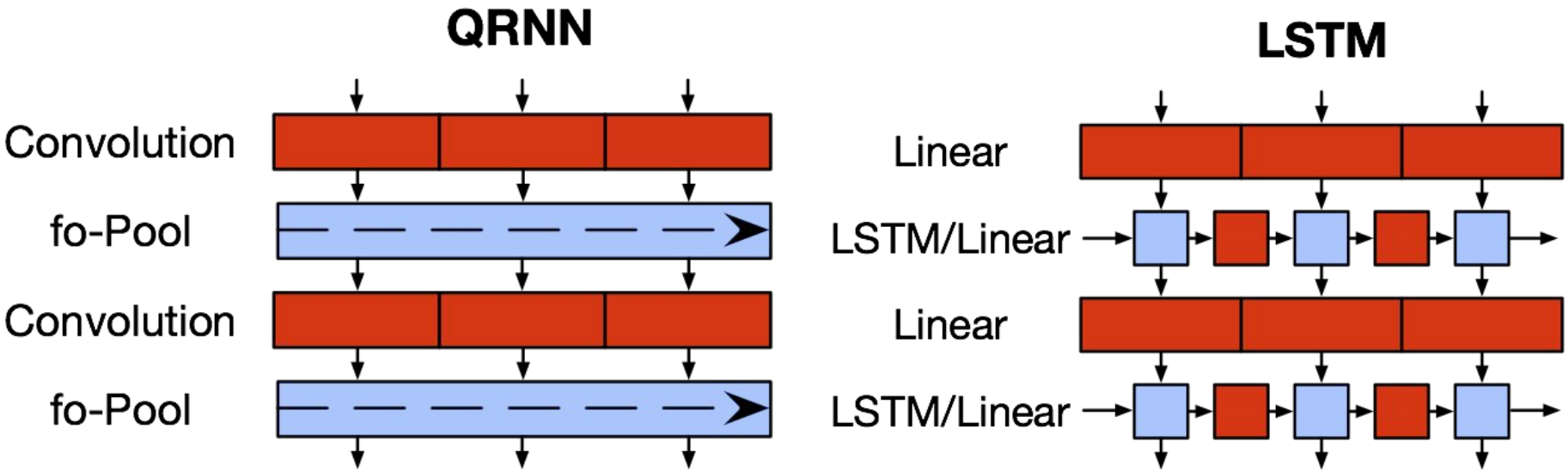
Long Short Term Memory (LSTM)

$$\mathbf{z}_t = \tanh(\mathbf{W}_z \mathbf{x}_t + \mathbf{V}_z \mathbf{h}_{t-1} + \mathbf{b}_z)$$
$$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{V}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$$
$$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{V}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$$
$$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{V}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$$
$$\mathbf{c}_t = \mathbf{i}_t \odot \mathbf{z}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$



Used with permission from Chris Olah's blog

RNN → QRNN


QRNN
Convolution
fo-Pool
Convolution
fo-Pool
LSTM
Linear
LSTM/Linear
Linear
LSTM/Linear

QRNN in detail

- Start with 1D convolution
  - no dependency on the hidden state
  - 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

$$\mathbf{z}_t = \tanh(\mathbf{W}_z * \mathbf{X} + \mathbf{b}_z)$$
$$[\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i * \mathbf{X} + \mathbf{b}_i)]$$
$$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f * \mathbf{X} + \mathbf{b}_f)$$
$$[\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o * \mathbf{X} + \mathbf{b}_o)]$$

f-pooling:

$$\mathbf{h}_t = (1 - \mathbf{f}_t) \odot \mathbf{z}_t + \mathbf{f}_t \odot \mathbf{h}_{t-1}$$

fo-pooling:

$$\mathbf{c}_t = (1 - \mathbf{f}_t) \odot \mathbf{z}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

ifo-pooling:

$$\mathbf{c}_t = \mathbf{i}_t \odot \mathbf{z}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

QRNN in detail

- 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

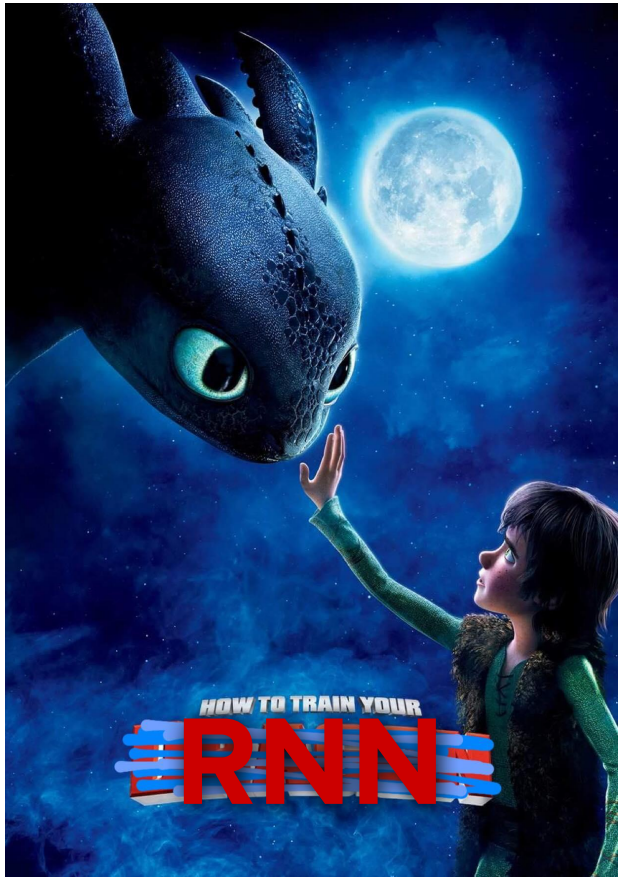
Codebase for PyTorch QRNN:
https://github.com/salesforce/pytorch-qrnn

```
extern "C" __global__ void qrnn_pool_fwd(
        const CArray<float, 3> f, const CArray<float, 3> z,
        CArray<float, 3> h) {
    int index[3];
    const int t_size = f.shape()[1];
    index[0] = blockIdx.x;
    index[1] = 0;
    index[2] = blockIdx.y * THREADS_PER_BLOCK + threadIdx.x;
    float prev_h = h[index];
    for (int i = 0; i < t_size; i++){
        index[1] = i;
        const float ft = f[index];
        const float zt = z[index];
        index[1] = i + 1;
        float &ht = h[index];
        prev_h = prev_h * ft + zt;
        ht = prev_h;
    }
}
extern "C" __global__ void qrnn_pool_bwd(
    const CArray<float, 3> f, const CArray<float, 3> gh,
    CArray<float, 3> gz) {
    int index[3];
    const int t_size = f.shape()[1];
    index[0] = blockIdx.x;
    index[2] = blockIdx.y * THREADS_PER_BLOCK + threadIdx.x;
    index[1] = t_size - 1;
    float &gz_last = gz[index];
    gz_last = gh[index];
    float prev_gz = gz_last;
    for (int i = t_size - 1; i > 0; i--){
        index[1] = i;
        const float ft = f[index];
        index[1] = i - 1;
        const float ght = gh[index];
        float &gzt = gz[index];
        prev_gz = prev_gz * ft + ght;
        gzt = prev_gz;
    }
}
```

Regularization for training an RNN

- **Standard dropout** on input, output
- **Recurrent dropout** between h_t and h_{t+1}
  (our preferred: weight dropped RNN)
- **Activation regularization**
  (add a loss for large (L2) outputs)
- **Temporal activation regularization**
  (penalize quick changes between hidden states)

Regularizing and Optimizing LSTM Language Models
https://arxiv.org/abs/1708.02182

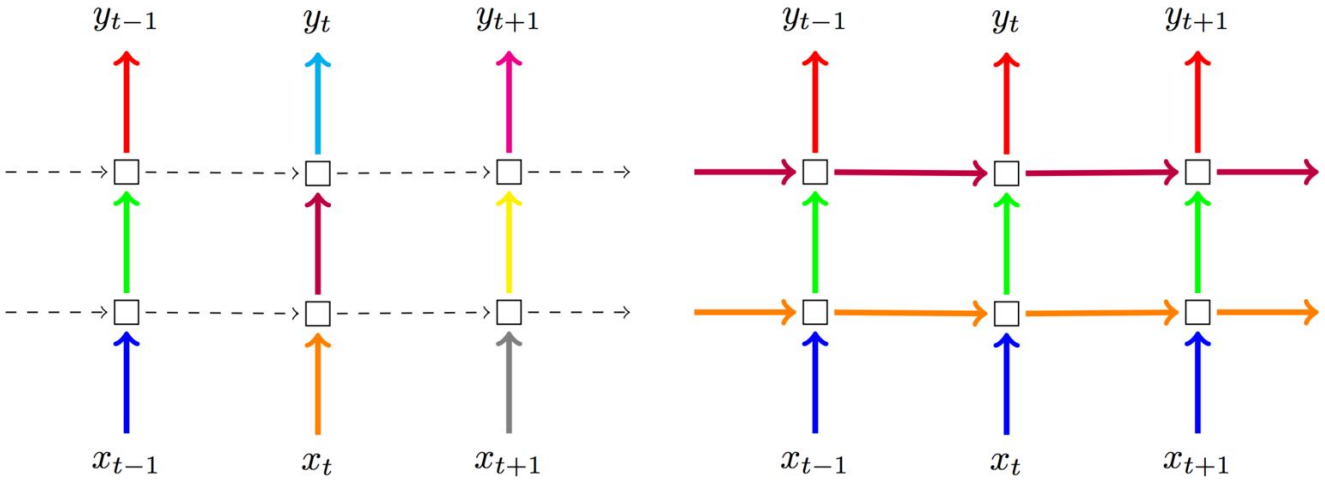


Recurrent Dropout

Almost nowhere in modern networks do we avoid adding dropout

… so why do we avoid placing dropout on recurrent connections?

(note: QRNN needs minimal recurrent dropout as it has a simple recurrence!)

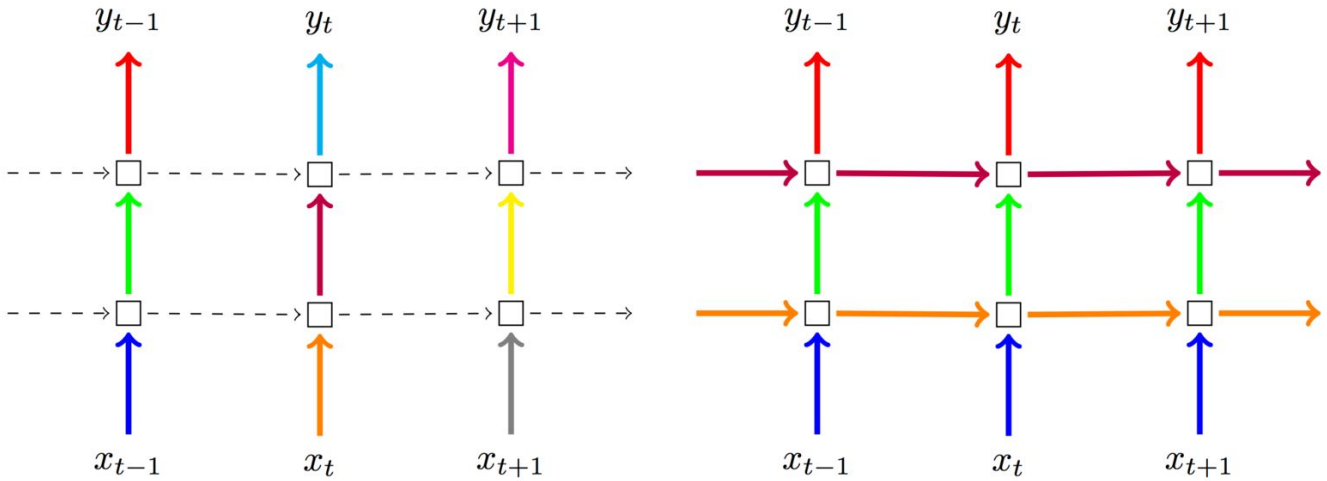


(a) Naive dropout RNN

(b) Variational RNN

Weight Dropped RNN

Our technique allows for recurrent dropout without modifying a blackbox LSTM:
- DropConnect (dropout on weight matrices) is applied to recurrent matrices
- The same neurons are inhibited the same way for each timestep



(a) Naive dropout RNN

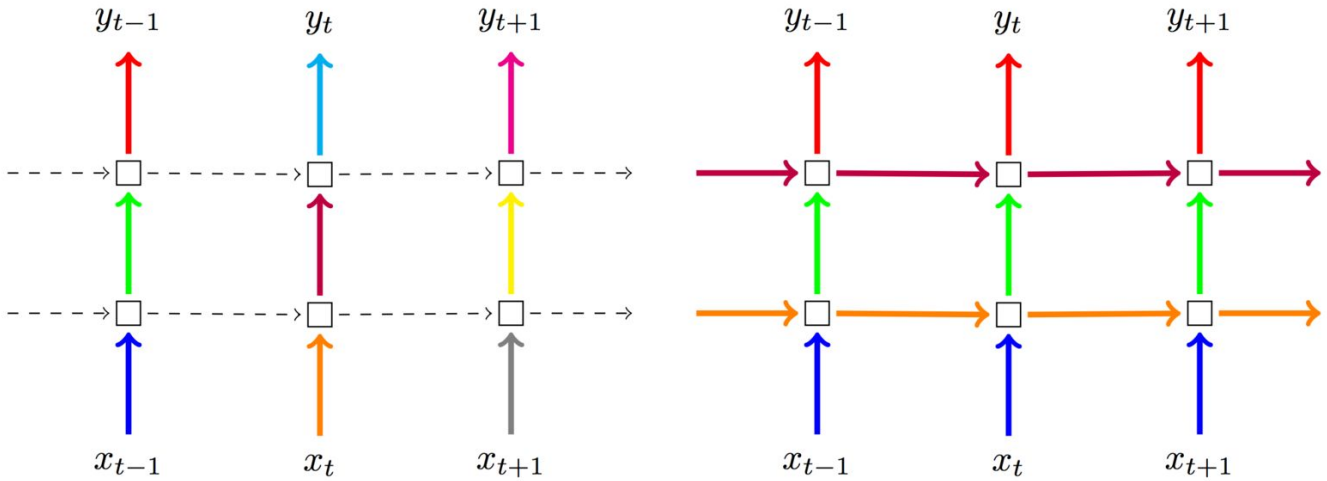
(b) Variational RNN

Weight Dropped RNN

Our technique allows for recurrent dropout *without modifying a blackbox LSTM*

This means fully compatible with NVIDIA cuDNN's optimized LSTM :)

For PyTorch code, see https://github.com/salesforce/awd-lstm-lm



(a) Naive dropout RNN

(b) Variational RNN

Softmax

For word level models with a large vocabulary, the softmax is:

- The majority of your model's parameters
- Slow to compute (linear in size of the vocabulary)

Softmax → Tied Softmax

For word level models with a large vocabulary, the softmax is:

- ~~The majority of your model's parameters~~
- Slow to compute (linear in size of the vocabulary)

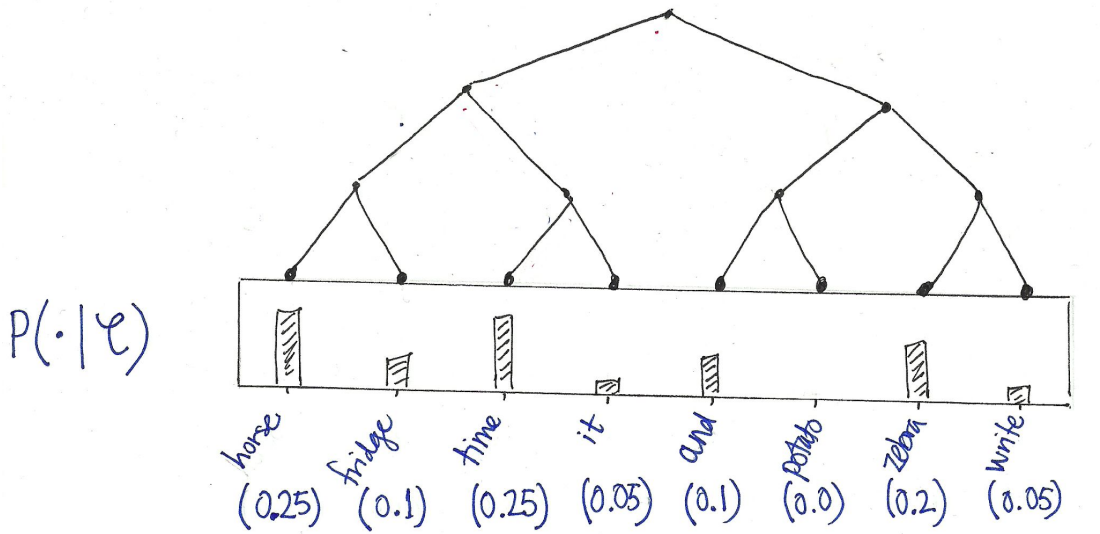
The tied softmax (Inan et al. 2016; Press & Wolf, 2016) re-uses the embedding's word vectors for the softmax weights, meaning:

- Essentially halves the number of parameters for larger models
- Training is faster *and* better

Softmax → Hierarchical Softmax

By placing a tree over the vocabulary, it's now **O(log N)** vs **O(N)**

Issue: Inefficiently uses the GPU during training

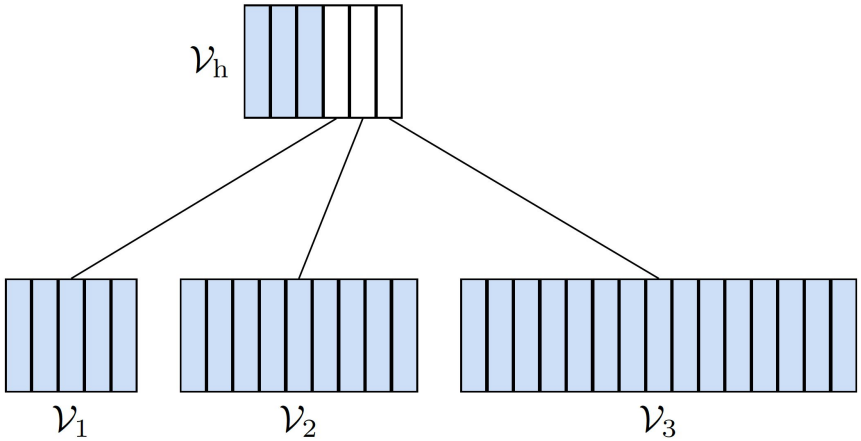


From Benjam Wilson's Hierarchical Softmax

Softmax → Adaptive Softmax (Grave et al. 2016)

Minimize the N-ary tree's height and "load balance" it:

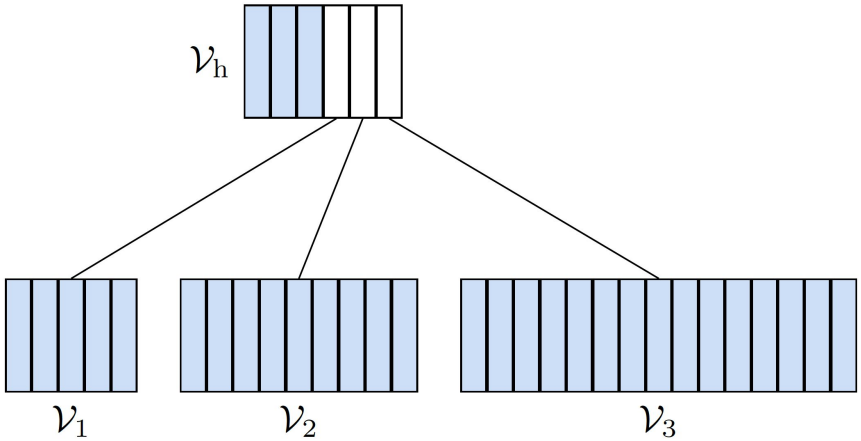
- The most frequent words (shortlist) appear in in the highest softmax
- The tree is only allowed to be of height two
- The clusters are organized such that each softmax's compute is GPU optimal



Softmax → Tied Adaptive Softmax

For word level models with a large vocabulary:

- Adaptive softmax approximation can impact accuracy (but aims to minimize that)
- Effectively utilizes the GPU
- Essentially halves the number of parameters for larger models
- Training is faster *and* better

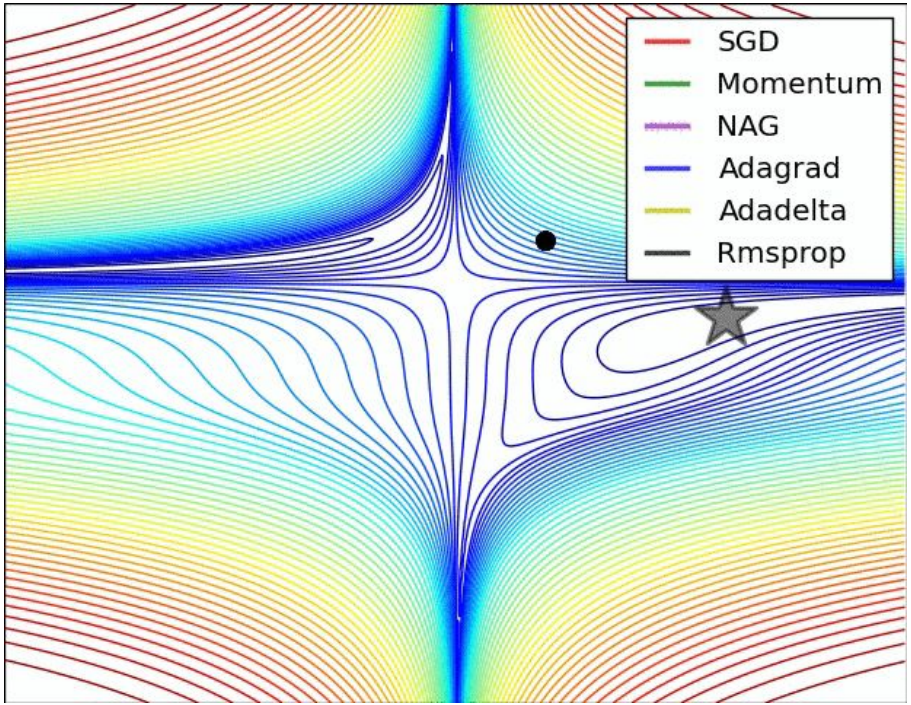


Training Strategy

✔ Model
✔ Gradient (through backprop)
Now what?

Many options, each with several hyperparameters:
- SGD + Momentum
- Adam
- RMSProp
- AMSGrad
- Adagrad
- Adadelta

…



SGD and Adam in a nutshell

SGD

$w_{t+1} = w_t - \alpha\, g_t$

Pick/Tune α, reduce based on condition(s)

Also used with momentum (i.e., $ß(w_t - w_{t-1})$)

Adam

$w_{t+1} = w_t - \alpha\, (cRMS(g_t)/cRMS(g_t^2))$

cRMS: Bias-corrected RMS mean.

Pick/Tune α, reduce based on condition(s)

Adam is great - but buyer beware.

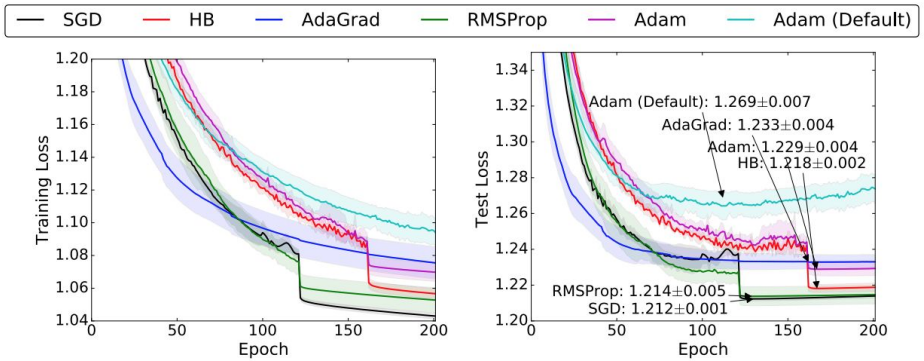
Typically:

- Usually works well with default (or close-to-default) hyperparameters
- Typically, very fast convergence
- But, ***might*** generalize worse

The Marginal Value of Adaptive Gradient Methods in Machine Learning

Ashia C. Wilson♯, Rebecca Roelofs♯, Mitchell Stern♯,
Nathan Srebro†, and Benjamin Recht♯*

♯ University of California, Berkeley.
† Toyota Technological Institute at Chicago



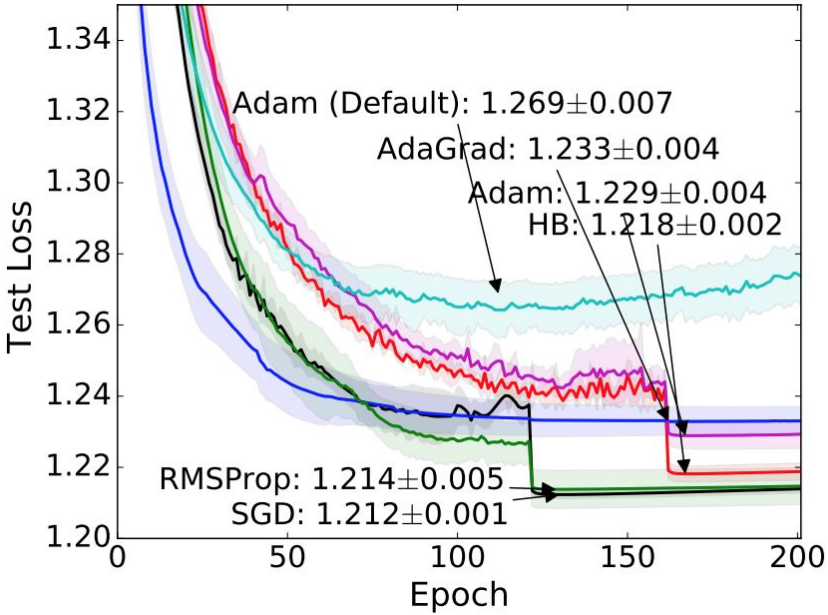
SGD is “the best”, if you’re willing to tune it.

SGD enables:

- Best generalization (theoretically and circumstantially, empirically)
- Lower memory requirements
- Easier parallelism

At the cost of:

- More difficult tuning (absence of bounds)
- Slower initial convergence

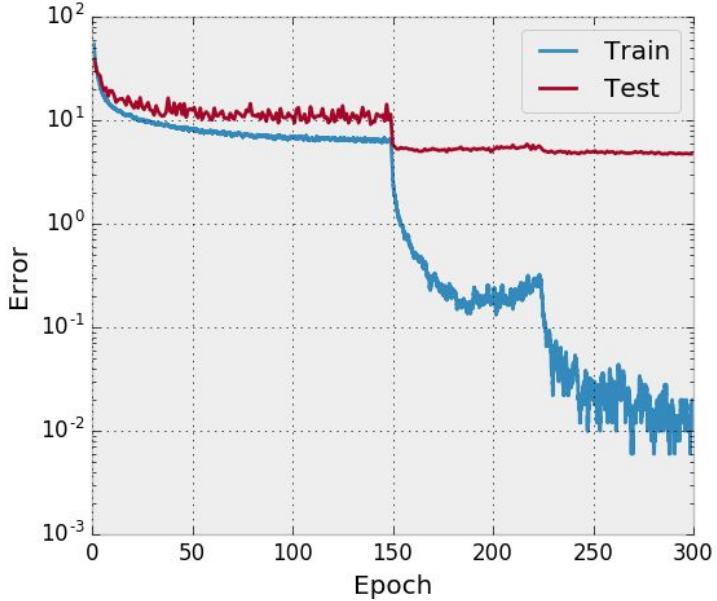


Tuning the learning rate is just half the story

The schedule is as (if not more) important!

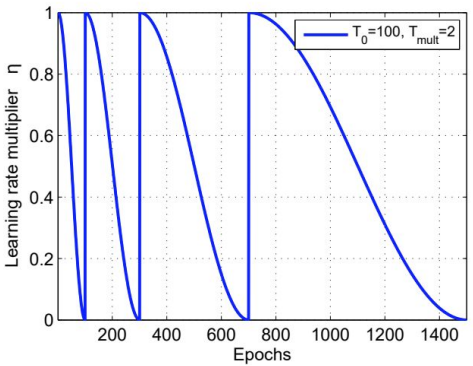
Typically:

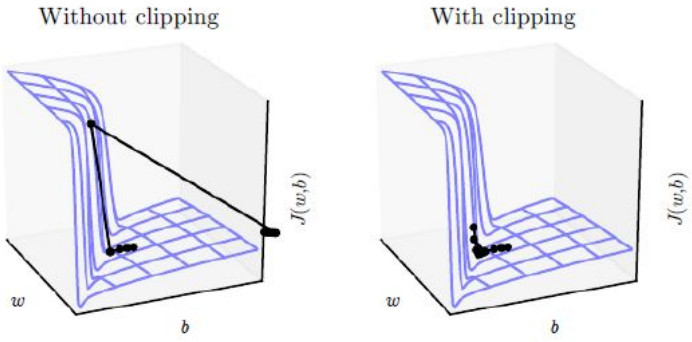
- Reducing by 10 (or 2) is better than a linear decay
- Too early? Irreversible bad decision. Too late? Waste of epochs.
- Can decide based on validation set
- Use a fixed-time scheduling, like 50-75
- If using large batch sizes, ramp up the learning rate to a larger value. Then reduce as above.

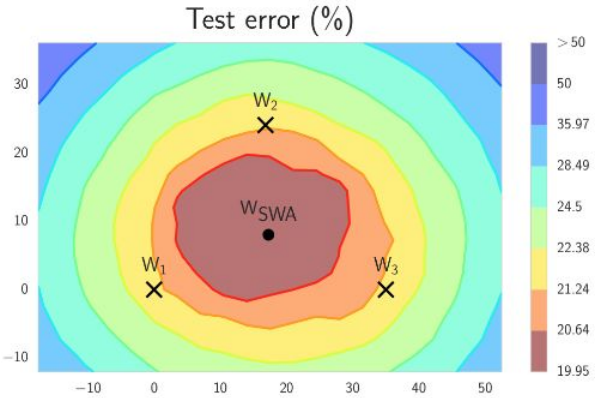


Additional heuristics might give last bit of performance

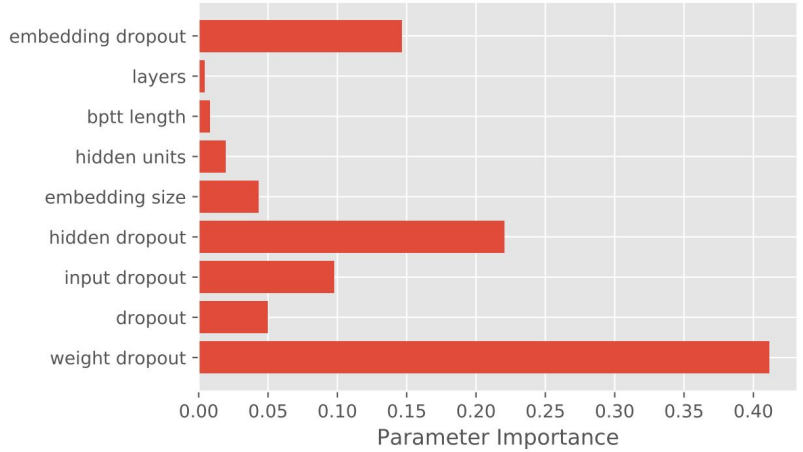
- Cyclic/cosine learning rates
- Weight averaging
- Gradient clipping
- Other optimizers (Adadelta, RMSprop, …)

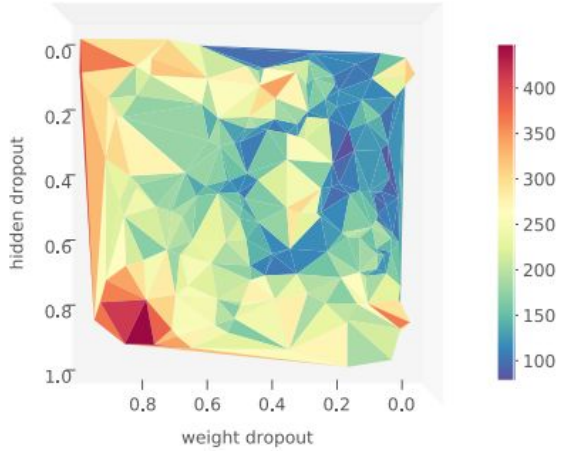




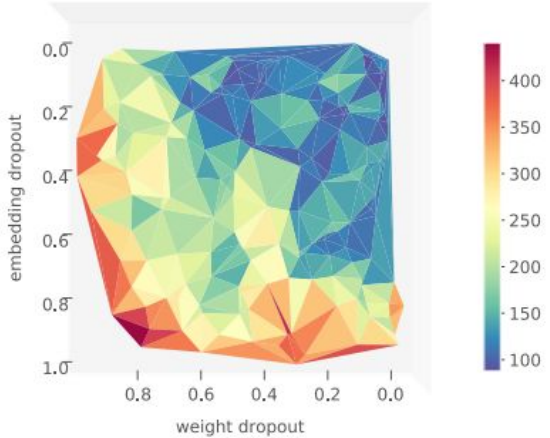


Analysis of Hyperparameters

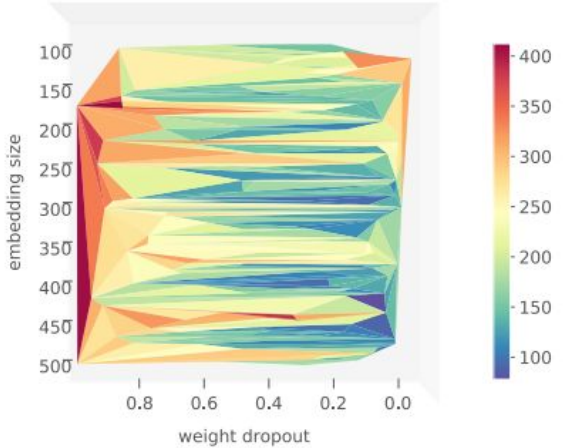




(a) Joint influence of weight dropout and hidden-to-hidden dropout.



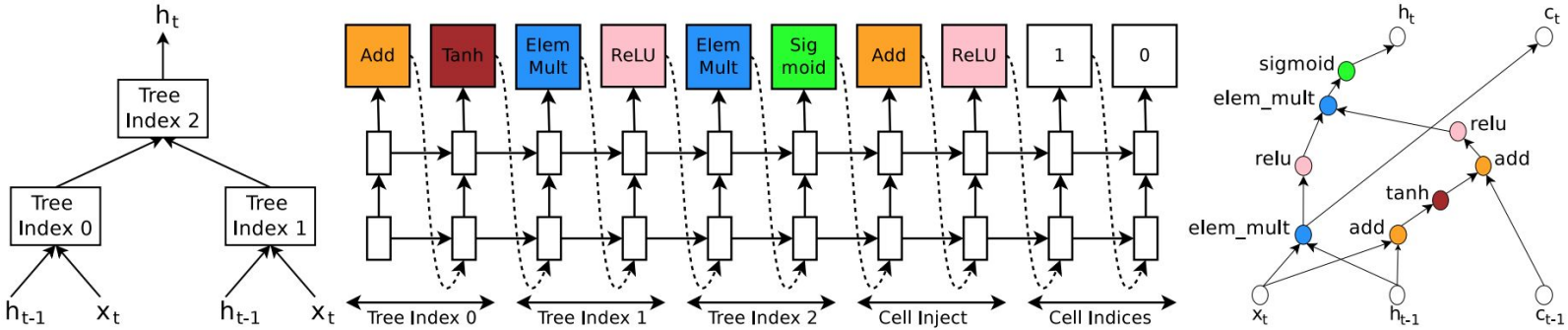
(b) Joint influence of weight dropout and embedding dropout.



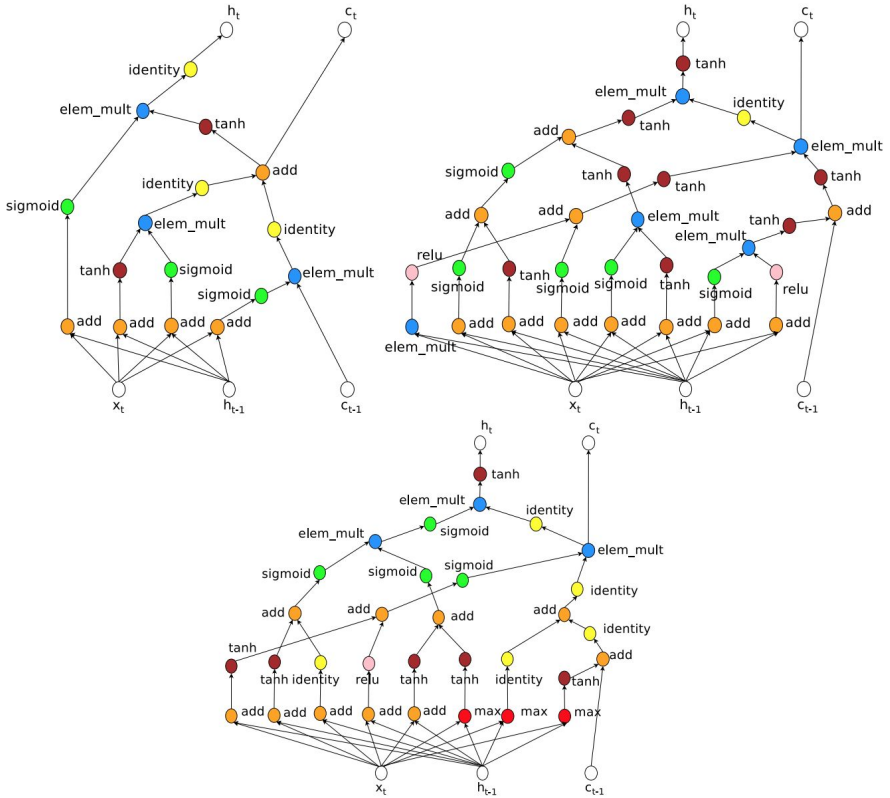
(c) Joint influence of weight dropout and embedding size.

Neural Architecture Search

- LSTMs are generic architectures for sequence modeling, why not customize?
- Create a reinforcement learning agent that *proposes* architecture and receives (validation) reward
- Profit!



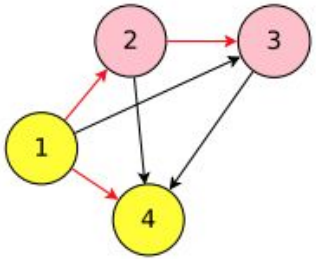
NASCell and BC3 Cell - Expensive!

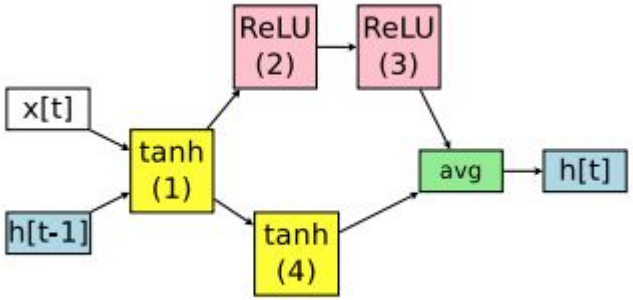


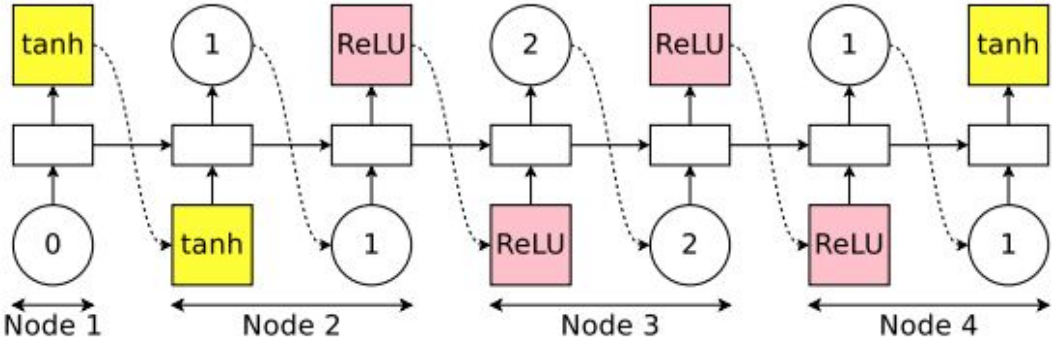
DSL BC3 Definition

```
Gate3(
    Tanh(
        Gate3(
            MM(x_t),
            Mult(
                MM(
                    Mult(MM(c_{t-1}),MM(x_t))
                ),
                MM(x_t)
            ),
            Sigmoid(
                Add( MM(x_t), MM(h_{t-1}) )
            )
        )
    ),
    h_{t-1},
    Sigmoid(
        Add( MM(x_t), MM(h_{t-1}) )
    )
)|12
```

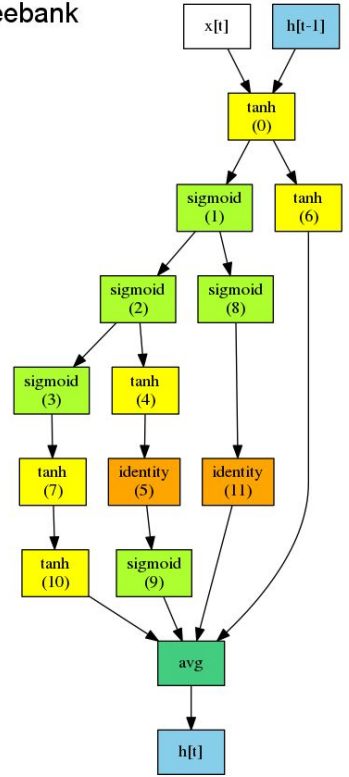
Current SOTA Approach - ENAS







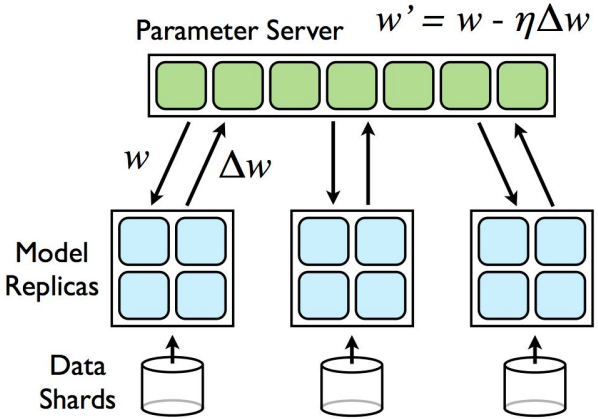
Penn Treebank
step: 50



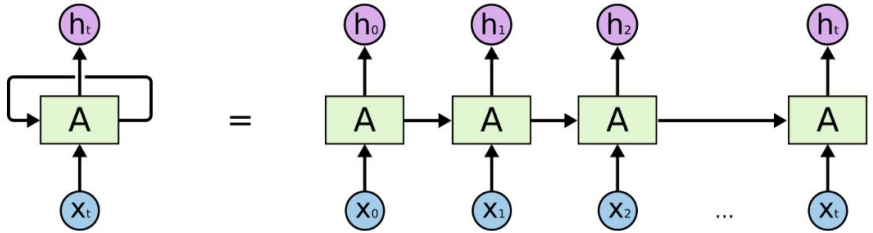
Results: Running on a single NVIDIA GTX 1080TI, ENAS finds a recurrent cell in about 10 hours

Parallelization - Batch Size and BPTT Length

- Increase batch size; embarrassingly parallel. Typically, synchronous
- Adjust learning rate accordingly
- Different variants depending on topology



- Language modeling is unique; you get to *pick* your sequence length
- High concurrency through larger BPTT lengths for QRNN-like architectures
- If training still stable, win-win! Better long-term dependency capturing and parallelization.



… but at the end of the day, **data is key**

- How much data do you have?
    - The dataset size can substantially change how you perform regularization

- Sentence level or paragraph level?
    - Do you need or want long term dependencies?

- For tokens, at what granularity are you looking at them?
  [Character, Subword, Word]
    - For word level, how do you handle OoV?

… but at the end of the day, **data is key**

The best algorithm in the world will still fail with bad data …
even with good data if the data is presented poorly!

Example: Standard BPTT length was always 35 tokens per batch
… but this means your model only ever sees data in the same position!

BPTT length should be randomized to ensure data is seen with different contexts

Summary

- **Understand your data**. What assumptions does it make? What assumptions are you making about it? Are you presenting it in a coherent way to your model?

- **Start with a baseline model**, use educated guesses for hyperparameters
  This baseline should be **fast** and **well tuned** = testbed for rapid experimentation

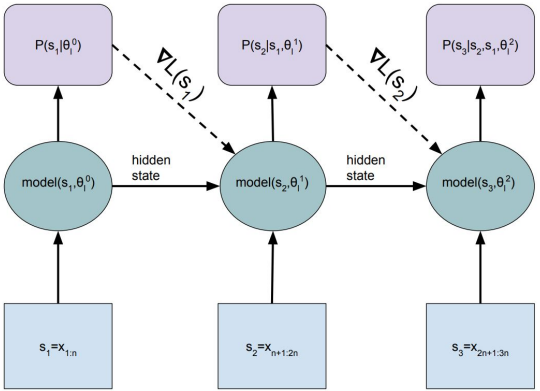
- Take **deliberate** and **reasoned** steps towards more complex models

- Unless you have strong proof that it's necessary, **don't sacrifice speed**

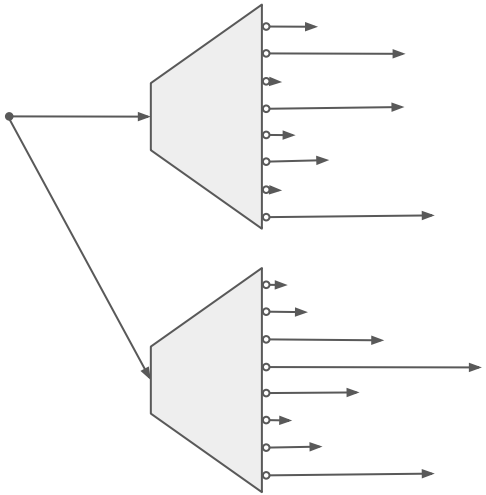
Cherry on top!

The benefits of open sourcing your work: smart people build on it :)

Dynamic Evaluation
(Krause et al. 2017)



Mixture of Softmaxes
(Yang et al. 2017)



Open Research Questions

- Neural language models are much better than traditional approaches but orders-of-magnitude slower. Latency issues in speech recognition.
- Understanding *how* the model learns to model language and why certain choices work or don't work.
- Better metrics for language modeling? Exact Match (EM), top-K accuracy?
- Understanding and improving error modes such as:
  - Arithmetic: "Australia and India are separated by 7800 km, which in miles is <N>"
  - Generation: Issues of repetition, entailment, coherence, long term dependencies / choices.
  - Context switches.
- Generalize to different *templates* or *styles* of language.
- Character vs BPE vs subword vs word

Open Sourcing

Regularizing and Optimizing LSTM Language Models
https://arxiv.org/abs/1708.02182

An Analysis of Neural Language Modeling at Multiple Scales
https://arxiv.org/abs/1803.08240

Codebase for AWD-LSTM and FastLM:
https://github.com/salesforce/awd-lstm-lm

Codebase for PyTorch QRNN:
https://github.com/salesforce/pytorch-qrnn

State-of-the-Art Large Scale Language Modeling in 12 Hours With a Single GPU

Nitish Shirish Keskar - @strongduality
Stephen Merity - @smerity