Neural Networks

Part 2

Normalization Speedups and Processing Sequential Data

Jon Dehdari and Asad Sayeed

January 18, 2017

Good Morning!

First things first

- Just want to know: how well do we remember our matrix multiplication?

First things first

- Just want to know: how well do we remember our matrix multiplication?
- Or linear algebra?

First things first

- Just want to know: how well do we remember our matrix multiplication?
- Or linear algebra?
- Then when we intone this:

$$\tanh(W\mathbf{x} + \mathbf{b})$$

  We all get what it means?

First things first

- Just want to know: how well do we remember our matrix multiplication?
- Or linear algebra?
- Then when we intone this:

  $$\tanh(W\mathbf{x} + \mathbf{b})$$

  We all get what it means?
- And do we remember what regression actually is?

First things first

- Just want to know: how well do we remember our matrix multiplication?
- Or linear algebra?
- Then when we intone this:

  $$\tanh(W\mathbf{x} + \mathbf{b})$$

  We all get what it means?
- And do we remember what regression actually is?
- When I say "train a model", do we all know what training an NN really "means"?

First things first

- Just want to know: how well do we remember our matrix multiplication?
- Or linear algebra?
- Then when we intone this:

$$\tanh(W\mathbf{x} + \mathbf{b})$$

  We all get what it means?
- And do we remember what regression actually is?
- When I say "train a model", do we all know what training an NN really "means"?

OK, great, I just wanted to make sure we're on the same page

Second things second

- OK, now that we have *that* out of the way, remember softmax?

$$P(y|\mathbf{x}) = \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{Z}$$

← exponentiation helps ensure scores are positive

← **normalization constant**, to ensure the score of all possible outcomes sums to 1

$$= \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{\sum_h e^{\boldsymbol{W}_h \cdot \mathbf{x}}}$$

← to get Z, we just add up scores from all possible outcomes

$$= \text{softmax}(\boldsymbol{W}_y \cdot \mathbf{x})$$

Second things second

- OK, now that we have *that* out of the way, remember softmax?

$$P(y|\mathbf{x}) = \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{Z}$$

← exponentiation helps ensure scores are positive

← **normalization constant**, to ensure the score of all possible outcomes sums to 1

$$= \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{\sum_h e^{\boldsymbol{W}_h \cdot \mathbf{x}}}$$

← to get Z, we just add up scores from all possible outcomes

$$= \text{softmax}(\boldsymbol{W}_y \cdot \mathbf{x})$$

- What does it mean?

Second things second

- OK, now that we have *that* out of the way, remember softmax?

$$P(y|\mathbf{x}) = \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{Z}$$

$\leftarrow$ exponentiation helps ensure scores are positive

$\leftarrow$ **normalization constant**, to ensure the score of all possible outcomes sums to 1

$$= \frac{e^{\boldsymbol{W}_y \cdot \mathbf{x}}}{\sum_h e^{\boldsymbol{W}_h \cdot \mathbf{x}}}$$

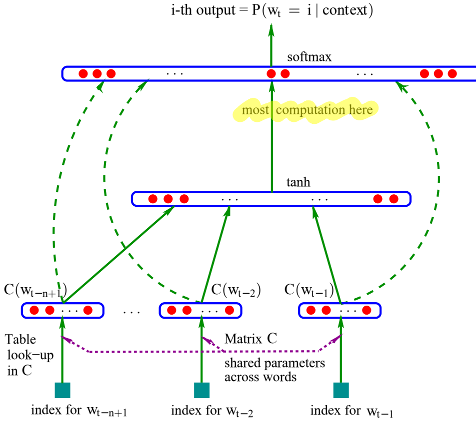
$\leftarrow$ to get Z, we just add up scores from all possible outcomes

$$= \text{softmax}(\boldsymbol{W}_y \cdot \mathbf{x})$$

- What does it mean?
- But there's a problem...

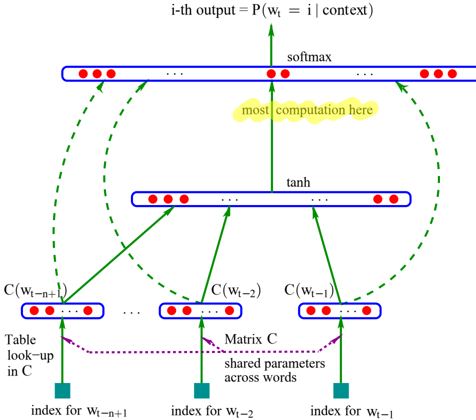
Softmax Normalization

- The slowest part of training a neural net LM is softmax normalization
- Why? Before the softmax layer (final layer) we just have a real number, not a probability
- So we need to know the sum of scores for all possible words being predicted (ie. the normalization constant)



Softmax Normalization

- The slowest part of training a neural net LM is softmax normalization
- Why? Before the softmax layer (final layer) we just have a real number, not a probability
- So we need to know the sum of scores for all possible words being predicted (ie. the normalization constant)



- This involves $|V|$ steps, where $|V|$ is the size of the vocabulary
- Typical values of $|V|$ are between 10K to 10M
- We must do this for every word in our training set (eg. 1M–1B), every epoch (> 10)

Speeding Up Normalization

- Can we speed up normalization? We can approximate Z:

Speeding Up Normalization

- Can we speed up normalization? We can approximate Z:
- **Class-based Decomposition** works like class-based LMs: first determine prob. of a given word's class/POS, then the prob. of the specific word $\mathcal{O}(\sqrt{|V|})$

Speeding Up Normalization

- Can we speed up normalization? We can approximate Z:
- **Class-based Decomposition** works like class-based LMs: first determine prob. of a given word's class/POS, then the prob. of the specific word $\mathcal{O}(\sqrt{|V|})$
- **Hierarchical Softmax** extends this idea to a fully binary-branching hierarchy of the vocabulary (like an ontology) $\mathcal{O}(\log_2(|V|))$

Speeding Up Normalization

- Can we speed up normalization? We can approximate Z:
- **Class-based Decomposition** works like class-based LMs: first determine prob. of a given word's class/POS, then the prob. of the specific word $\mathcal{O}(\sqrt{|V|})$
- **Hierarchical Softmax** extends this idea to a fully binary-branching hierarchy of the vocabulary (like an ontology) $\mathcal{O}(\log_2(|V|))$
- **Noise Contrastive Estimation** (NCE) disposes with MLE (in Softmax). Instead, a binary classifier is learned: observed training data vs. artificially generated noise. word2vec's negative sampling is a simplified version. $\mathcal{O}(1)$

Speeding Up Normalization

- Can we speed up normalization? We can approximate Z:
- **Class-based Decomposition** works like class-based LMs: first determine prob. of a given word's class/POS, then the prob. of the specific word $\mathcal{O}(\sqrt{|V|})$
- **Hierarchical Softmax** extends this idea to a fully binary-branching hierarchy of the vocabulary (like an ontology) $\mathcal{O}(\log_2(|V|))$
- **Noise Contrastive Estimation** (NCE) disposes with MLE (in Softmax). Instead, a binary classifier is learned: observed training data vs. artificially generated noise. word2vec's negative sampling is a simplified version. $\mathcal{O}(1)$
- **Self Normalization** ensures that the normalization constant Z is close to one. Slow for training, fast for test-time queries

Neural Networks for Sequential Data

- Feedforward (FF) networks only indirectly deal with sequential data (like language)
- FF Neural LMs are basically 'soft' *n*-gram LMs – their history is still fixed

Neural Networks for Sequential Data

- Feedforward (FF) networks only indirectly deal with sequential data (like language)
- FF Neural LMs are basically 'soft' *n*-gram LMs – their history is still fixed
- The model needs to 'remember' a longer history, with loops

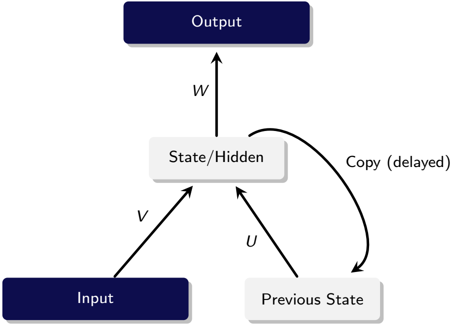
Recurrent Neural Networks

A neural net with loops is called **recurrent**

Recurrent Neural Networks

A neural net with loops is called **recurrent**

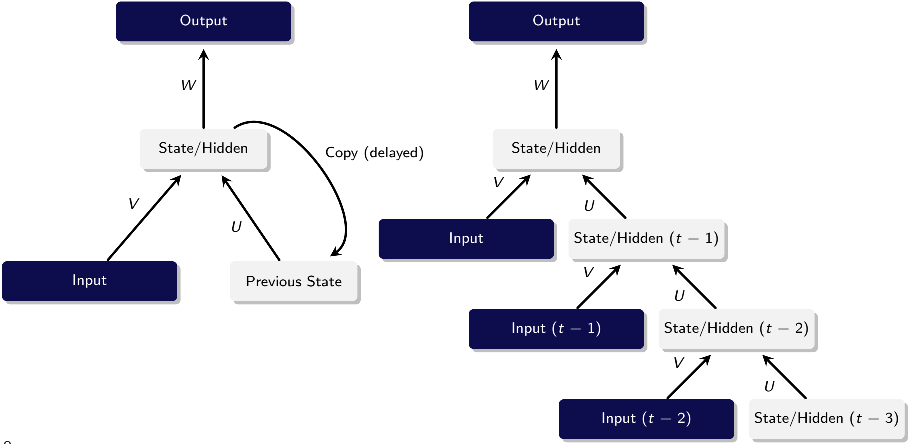
- The current hidden layer of the model is based on both the current word and the hidden layer of the previous timestep
- This is implemented by copying the hidden layer to another layer, overwriting the existing weights
- This specific RNN is called an **Elman network** (or **simple RNN** / SRN)



- To train an RNN, we first need to 'unroll' the loops

Training RNNs with BPTT

- Backpropagation through time (BPTT) trains RNNs by unrolling the most recent part of the loop
- Now the network is feedforward
- Below is an example of an unrolled RNN using last 3 states $(\tau = 3)$



Problems with Elman Networks / SRNs

- The main problem with Elman networks (SRNs) is that gradients less than 1 become exponentially small over time (the **vanishing gradient problem**) ...
- and gradients greater than 1 become exponentially large over time (the **exploding gradient problem**)*
- This leads to instability, and bad results

Problems with Elman Networks / SRNs

- The main problem with Elman networks (SRNs) is that gradients less than 1 become exponentially small over time (the **vanishing gradient problem**) ...
- and gradients greater than 1 become exponentially large over time (the **exploding gradient problem**)*
- This leads to instability, and bad results
- What if we had another neural network help the first network learn long-distance relationships?

Problems with Elman Networks / SRNs

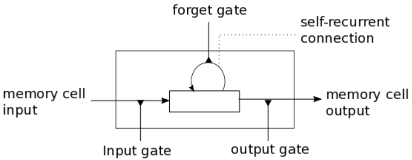
- The main problem with Elman networks (SRNs) is that gradients less than 1 become exponentially small over time (the **vanishing gradient problem**) ...
- and gradients greater than 1 become exponentially large over time (the **exploding gradient problem**)*
- This leads to instability, and bad results
- What if we had another neural network help the first network learn long-distance relationships?
- That's basically what we do when we add more weight matrices to a neural network

Problems with Elman Networks / SRNs

- The main problem with Elman networks (SRNs) is that gradients less than 1 become exponentially small over time (the **vanishing gradient problem**) ...
- and gradients greater than 1 become exponentially large over time (the **exploding gradient problem**)*
- This leads to instability, and bad results
- What if we had another neural network help the first network learn long-distance relationships?
- That's basically what we do when we add more weight matrices to a neural network
- As you might guess, that's what we're going to do ...

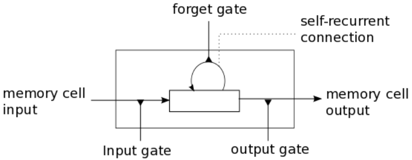
* The exploding gradient problem can be alleviated by clipping large gradient values to some maximum number.

Long Short-term Memory



- A **long short-term memory** (LSTM) network adds more weight matrices to function as *soft* 'memory gates', so that long-distance phenomena in our data can be held in the network over multiple timesteps

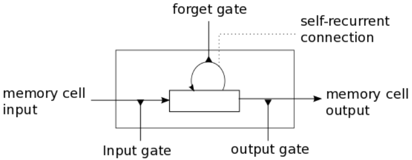
Long Short-term Memory



- A **long short-term memory** (LSTM) network adds more weight matrices to function as *soft* 'memory gates', so that long-distance phenomena in our data can be held in the network over multiple timesteps
- Input gate: $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
- Candidate memory state: $\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$
- Forget gate: $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
- Memory state: $C_t = i_t \odot \tilde{C}_t + f_t \odot \tilde{C}_{t-1}$
- Output gate: $o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$
- Output: $h_t = o_t \odot \tanh(C_t)$

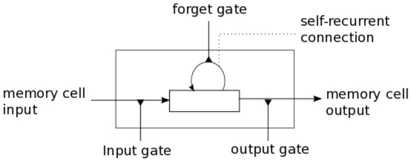
Image courtesy of a nice tutorial at `http://deeplearning.net/tutorial/lstm.html`. Another nice tutorial is at `https://colah.github.io/posts/2015-08-Understanding-LSTMs`. The symbol $\odot$ is the Hadamard product (a.k.a. elementwise multiplication), which just multiplies corresponding elements of two matrices and returns another matrix of their products.

LSTM - anatomy



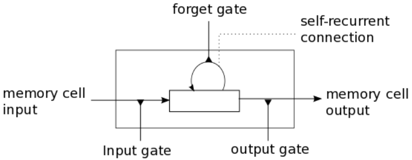
- Input gate: $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
  - Sigmoid - between 0 and 1, used to weight importance of input.
  - x_t current input, h_{t-1} previous hidden state.

LSTM - anatomy



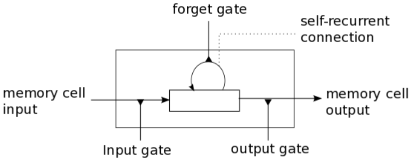
- Input gate: $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
  - Sigmoid - between 0 and 1, used to weight importance of input.
  - x_t current input, h_{t-1} previous hidden state.
- Candidate memory state: $\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$
  - What the cell should remember given current input and previous hidden state, other things being equal
  - tanh makes it between -1 and 1.

LSTM - anatomy



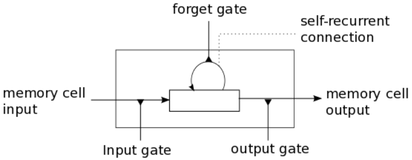
- Input gate: $i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
  - Sigmoid - between 0 and 1, used to weight importance of input.
  - x_t current input, h_{t-1} previous hidden state.
- Candidate memory state: $\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$
  - What the cell should remember given current input and previous hidden state, other things being equal
  - tanh makes it between -1 and 1.
- Now come the fun parts.

LSTM - anatomy



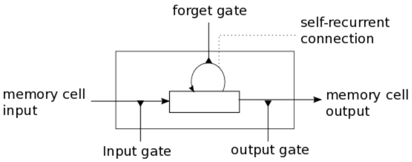
- Forget gate: $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
  - Sigmoid - between 0 and 1, used to weight how much we're going to discount current memory state.
  - x_t current input, h_{t-1} previous hidden state.
  - (Weights have to be learned for each gate type!)

LSTM - anatomy



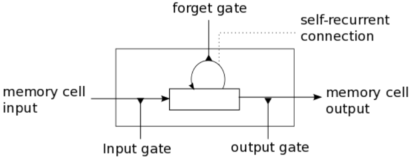
- Forget gate: $f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
  - Sigmoid - between 0 and 1, used to weight how much we're going to discount current memory state.
  - x_t current input, h_{t-1} previous hidden state.
  - (Weights have to be learned for each gate type!)
- Memory state: $C_t = i_t \odot \tilde{C}_t + f_t \odot \tilde{C}_{t-1}$
  - What the cell is going to remember, given the discount of the forget gate of the previous memory state.

LSTM - anatomy



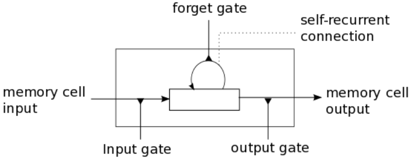
- Output gate: $o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$
  - Now includes weights not only on x_t and h_{t-1}, but also C_t – ie, how much the new memory state contributes to the next cell.

LSTM - anatomy



- Output gate: $o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$
  - Now includes weights not only on x_t and h_{t-1}, but also C_t – ie, how much the new memory state contributes to the next cell.
- Output: $h_t = o_t \odot \tanh(C_t)$ - then emit the new state for the next cell

LSTM - anatomy



- Output gate: $o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$
  - Now includes weights not only on x_t and h_{t-1}, but also C_t – ie, how much the new memory state contributes to the next cell.
- Output: $h_t = o_t \odot \tanh(C_t)$ - then emit the new state for the next cell

And then it's all a matter of training, simple, right?

Gates

- Each soft gate that we use is just another weight matrix that we apply to various inputs, then squash through a logistic function

Gates

- Each soft gate that we use is just another weight matrix that we apply to various inputs, then squash through a logistic function
- For example, if the value of the forget gate f is 0, the memory state from the previous timestep (C_{t-1}) is completely forgotten

Gates

- Each soft gate that we use is just another weight matrix that we apply to various inputs, then squash through a logistic function
- For example, if the value of the forget gate f is 0, the memory state from the previous timestep (C_{t-1}) is completely forgotten
- If $f = 1$, we fully keep the memory state of the previous timestep

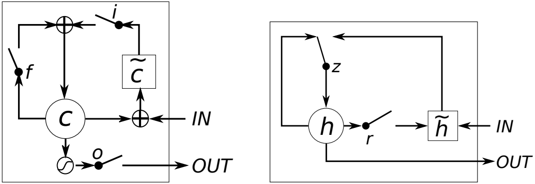
Gates

- Each soft gate that we use is just another weight matrix that we apply to various inputs, then squash through a logistic function
- For example, if the value of the forget gate f is 0, the memory state from the previous timestep (C_{t-1}) is completely forgotten
- If $f = 1$, we fully keep the memory state of the previous timestep
- The value of f can be between 0 and 1, so the memory decays

Gates

- Each soft gate that we use is just another weight matrix that we apply to various inputs, then squash through a logistic function
- For example, if the value of the forget gate f is 0, the memory state from the previous timestep (C_{t-1}) is completely forgotten
- If $f = 1$, we fully keep the memory state of the previous timestep
- The value of f can be between 0 and 1, so the memory decays
- That's a big difference over Elman networks / SRNs

Gated Recurrent Units (GRUs)



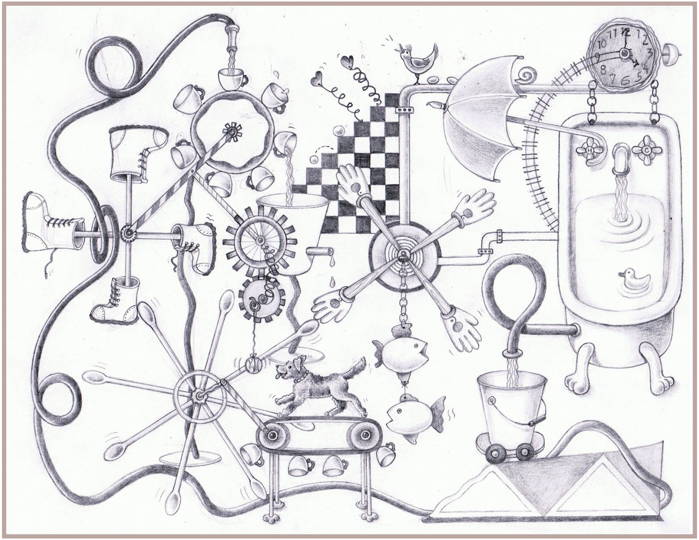
(a) Long Short-Term Memory (b) Gated Recurrent Unit

Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation.

- **Gated recurrent units** (GRUs) are very similar to LSTMs, but are a little simpler
- GRUs merge the forget and input gates into a single update gate
- GRUs also merge the hidden state and the cell state
- Both LSTMs and GRUs achieve similar performance on many tasks

Image courtesy of http://arxiv.org/pdf/1412.3555v1.pdf

Rube Goldberg Network



Further Reading

Overviews:

- https://www.reddit.com/r/MachineLearning/comments/44bxdj/scrn_vs_lstm/czp4hqr/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714
- http://deeplearning.net/tutorial/lstm.html
- http://arxiv.org/abs/1412.3555
- https://www.tensorflow.org/versions/master/tutorials/recurrent/index.html#recurrent-neural-networks
- http://keras.io/layers/recurrent/
- https://drive.google.com/open?id=0B-aFax-9-qt3Sllodkpmc1M0MUk
- https://en.wikipedia.org/wiki/LSTM

Original Papers:

- Elman, Jeffrey L. 1990. Finding Structure in Time. *Cognitive Science* 14.179–211.
- Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9.1735–1780.
- Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*.