Recurrent nets

Bryan Pardo
Deep Learning
Northwestern University
Dealing with time

• With a "standard" feed-forward architecture, you process data from within a window, ignoring everything outside the window.
• To get influence from the processing of earlier time steps, add nodes and connections
• This doesn’t scale well
Let’s look at that net.

- An entire new set of weights for EACH time step.
- Audio is sampled at 44,100 times per second
- The number of past time steps you could consider is limited by the architecture.
- The number of weights to learn quickly gets out of control.
Take an idea from CNNs and HMMs

- Markov property: The state of the world can be captured by knowing current state + immediately previous state
- Markov models use recurrent connections
- CNNs use the same set of shared weights on different parts of the input
Take an idea from CNNs and HMMs

- If all the windows share the same input weights (like in a feature map), then we only have the same number of weights as if we had a single window.
- This is a recurrent net.
- How do you train this?
- Are there any obvious limitations?
Backprop through time: “Unrolling”

- Pick a number of steps over which you’re going to “unroll” the net.
- Treat it like you’re training a convolutional neural net.
- Pick the number of steps based on your frenemy: Exponential decay.
Getting influence from the past: Skip connections (used in Highway networks)

- Widely used
- Limited by the length of the skip
Exponentially decaying influence

• If your network needs to connect information from a distant timestep, the influence of the earlier one tends to get lost
• Why? Exponential decay.
Exploding and vanishing gradients

- What if the weight on the recurrent connection is greater than 1?
- What if the weight on the recurrent link is less than 1?
- What if it is exactly 1?
An RNN example: Language modeling

• In language modeling, the game is to be able to predict the next word, given the previous N words.

• Examples
  “Two plus two equals...”
  “A stitch in time saves....”
  “I never did...”
Our text encoding

• 1000 most common English words. + start + stop + other

• Encoding: 1003 element one-hot vector for each word in a sentence
  • Word index determined by popularity
  • Start = 1001
  • Stop = 1002
  • Other (any word not in the top 1000) = 1003

• Examples:

  An apple is good for you. -> [1001, 48, 927, 24, 121, 7, 26, 1002]

  Lilliputian dilatants prognosticate parsimoniously! -> [1001, 1003, 1003, 1003, 1003, 1003, 1002]
The goal: predict the next token

• Each sentence is its own label.

• Given “An apple is…”, predict “good” as the next word.

• Our model output will be a probability distribution over the 1003 element vector (top 1000 words + start + stop + other).

• We can use cross-entropy loss, comparing the one-hot vector to the probability vector output by the model.
Our network

INPUT:

An apple is good for you

Argmax output

fruit is good

Probability distribution over words

4-node RNN
Hidden layer

INPUT:

An apple is good for you

t-2

t-1

t
A RNN with 4 hidden nodes: how many weights?

**Output layer**

Output a 1003 element probability distribution over the set of words. 

\[0.01, 0.0000098, \ldots, 0.0023, \ldots, 0.001, 0, 0.000053] \quad \text{Softmax activation}

**Hidden layer**

All hidden nodes are fully connected to the output layer.

**Input + prev state**

Input word \( w(t) \): a 1003 element one-hot vector encoding word \( t \).

\[0, 0, 0, \ldots, 1, \ldots, 0, 0, 0\]

Previous state \( s(t-1) \): a vector of the output from each hidden unit from time \( t-1 \).

\[0.2, 0.001, 0.3\]
RNN: the math

OUTPUT a 1003 element probability distribution over the set of words.

Output layer $[0.01, .0000098, \ldots, .0023, \ldots, .0010, .000053]$ Softmax activation

ALL HIDDEN NODES ARE FULLY CONNECTED TO THE OUTPUT LAYER

Hidden layer Sigmoid activation

Input + prev state

INPUT word $w(t)$: a 1003 element one-hot vector encoding word $t$.

Previous state $s(t-1)$: a vector of the output from each hidden unit from time $t-1$

$x(t) = [w(t), s(t-1)]$ this vector has $1003 + n$ elements

$w(t)$

$s(t-1) = [s_1(t-1), \ldots, s_n(t-1)]$
What if we use a more realistically sized net?

- Dictionary size = 50,000
- Hidden states = 100

- $50,000 \times 100 \times 2 = 10,000,000$

- It’s just that easy to have 10 million weights.

- Adding a couple of extra hidden layers (even fully connected ones) doesn’t cost you much, compared to the dictionary size.
RNN: the math

**INPUT word** $w(t)$: a 1003 element one-hot vector encoding word $t$.

**Previous state** $s(t-1)$: a vector of the output from each hidden unit from time $t-1$.

**Output layer** $[0.01, .0000098, ..., .0023, ..., .0010, .000053]$ Softmax activation

ALL HIDDEN NODES ARE FULLY CONNECTED TO THE OUTPUT LAYER

**Hidden layer**

**Input + prev state**

$x(t) = [w(t), s(t-1)]$ this vector has 1003 + $n$ elements

$w(t)$

$s(t-1) = [s_1(t-1), ..., s_n(t-1)]$

**OUTPUT** a 1003 element probability distribution over the set of words.
**RNN: the math**

**OUTPUT** a 1003 element probability distribution over the set of words.

Output layer \[ [0.01, 0.0000098, ..., 0.0023, ..., 0.0010, 0.000053] \] Softmax activation

\[ s_j(t) = \sigma \left( \sum_i u_{ij} x_i(t) \right) \]

\[ \sigma(z) = (1 + e^{-z})^{-1} \]

Hidden node activation function

\[ x(t) = [w(t), s(t - 1)] \] this vector has 1003 + \( n \) elements

\[ w(t) \quad s(t - 1) = [s_1(t - 1), ..., s_n(t - 1)] \]

Input + prev state

Each node \( j \)

INPUT word \( w(t) \): a 1003 element one-hot vector encoding word \( t \).

Previous state \( s(t - 1) \): a vector of the output from each hidden unit from time \( t-1 \)
RNN: the math

OUTPUT a 1003 element probability distribution over the set of words.

Each output $k$

$$y_k(t) = g \left( \sum_j s_j(t) v_{jk} \right)$$

$g(z_k) = \frac{e^{z_k}}{\sum_m e^{z_m}}$
Softmax

Each node $j$

$$s_j(t) = \sigma \left( \sum_i u_{ij} x_i(t) \right)$$

$\sigma(z) = (1 + e^{-z})^{-1}$
Hidden node activation function

$x(t) = [w(t), s(t - 1)]$
this vector has $1003 + n$ elements

$w(t)$

$s(t - 1) = [s_1(t - 1), ..., s_n(t - 1)]$
PREVIOUS STATE: an $n$ dimensional vector, where each element is the output of a hidden node at the previous time step $t-1$. 

INPUT word $w(t)$: a 1003 element one-hot vector encoding word $t$. 

Output layer

Hidden layer

Input + prev state
RNN: Predicting the next word

**prediction** = \( \hat{w}(t + 1) = \arg\max_k [y_1(t), ..., y_k(t), ..., y_m(t)] \)

Each output \( k \)

\[
y_k(t) = g \left( \sum_j s_j(t) v_{jk} \right)
\]

Each node \( j \)

\[
s_j(t) = \sigma \left( \sum_i u_{ij} x_i(t) \right)
\]

\[
x(t) = [w(t), s(t - 1)]
\]

\( w(t) \)

INPUT word \( w(t) \): a 1003 element one-hot vector encoding word \( t \).

\( s(t - 1) \)

PREVIOUS STATE: an \( n \) dimensional vector, where each element is the output of a hidden node at the previous time step \( t-1 \).

\( g(z_k) = \frac{e^{z_k}}{\sum_m e^{z_m}} \)

Softmax

\( \sigma(z) = (1 + e^{-z})^{-1} \)

Hidden node activation function

\( \sigma \) is the hidden node activation function, where \( \sigma(z) = \frac{1}{1 + e^{-z}} \)

\( \sum_{m} e^{z_m} \)

This vector has 1003 + \( n \) elements

Recurrent neural network-based language model (Interspeech 2010)
This is an autoregressive model

• An autoregressive model forecasts the variable of interest using a linear combination of past values of the variable

• The term autoregression indicates that it is a regression of the variable against itself
A language model is a generative model

• If you have something that predicts the next word, you have something that can “generate” the next word.

• Sentence completion is possible

• Sentence generation is possible
Generating a new sentence with the model

1. Pick an initialization token
2. Sample from the conditional distribution output by the model
3. Random initialization \([s_1(t-1), \ldots, s_n(t-1)]\)
4. Pick an initialization token

Example: She
Generating a new sentence with the model

She

from model output at previous time step

likes

Sample from the conditional distribution output by the model

Random initialization

$[s_1(t-1), \ldots, s_n(t-1)]$

START

from model output at previous time step

She
Generating a new sentence with the model

Sample from the conditional distribution output by the model

Random initialization

$[s_1(t-1), ..., s_n(t-1)]$

She

likes

cheese

START

She

likes
Generating a new sentence with the model

Sample from the conditional distribution output by the model

Random initialization $[s_1(t-1), \ldots, s_n(t-1)]$

Continue till you generate a stop token

START $\rightarrow$ She $\rightarrow$ likes $\rightarrow$ cheese $\rightarrow$ STOP
Perplexity

• A measure of how hard it is to guess the next word.
• The exponentiation of the cross-entropy

\[
Perplexity = 2^{H(p)} = 2^{- \sum_x p(x)(\log_2(q(x)))}
\]

• A commonly used measure of how well a language model is doing
• Measures how confused the model is (how many choices it has reduced the next word to)
Getting more context

• We predict/generate a new token, based on a prior sequence.

• Our generated output is contextually informed by the past

• But wait….if our training data is whole sentences, can’t we do the same thing from the ”future” (i.e. the next word or rest of sentence)?

• Sure we can. Just feed in the sequence backwards.
RNN: predicting the “past” based on the “future”

\[ \text{prediction} = \widehat{w}(t - 1) = \max_k [y_1(t), ..., y_k(t), ..., y_m(t)] \]

\[ y_k(t) = g \left( \sum_k s_j(t) v_{jk} \right) \]

\[ s_j(t) = \sigma \left( \sum_i u_{ij} x_i(t) \right) \]

\[ x(t) = w(t) + s(t + 1) \]

\[ w(t) \quad \quad \quad \quad s(t + 1) = [s_1(t + 1), ..., s_n(t + 1)] \]

INPUT word \( w(t) \): a 1003 element one-hot vector encoding word \( t \).
Bidirectional RNN

- Inform output layer’s probability distribution using a forward layer and a backwards layer

- The generated token(s) are influenced by both previous and subsequent context
Multi-layer RNN

• You can have multiple hidden layers, where layer n feeds into layer n+1

Long-Short Term Memories

Here’s a problem. What can learn to do it?

• X is a finite-length sequence composed of tokens, where each token $x_n \in \mathbb{R} \cup \{a, b\}$.

• The length of X is unknown.

• Before beginning, the total = 0.

• Iterate through X and do the following
  • If $x_n = a$, add $x_{n+1}$ to the total.
  • If $x_n = b$, return the total and reset the total to 0.
Let’s play

a, -1, a, 100, b = 99

1, 3, -5, a, 5, -1, 8 , 2, 0, a, 9, b = 14

1, 3, a, -5, 0, 0, 0, 0, a, 5, -1, 7 ,2, 0, a, 9, -12, b = 9
Feed-forward: Fixed-length time window

- If your network needs to connect information from outside the window, you lose.
RNN: exponentially decaying influence

- If your network needs to connect information from a distant timestep, the influence of the earlier one tends to get lost.
- Why? Exponential decay.
Long Short Term Memory Units (LSTMs)

• Added a way of storing data over many time steps without decay
• Let networks to handle problems with long term dependencies

LSTM training

- Error is propagated indefinitely through its memory cell, the constant error carousel (CEC)
- Error flow back through the unit is truncated at the incoming weights.

An easy-to-follow-visual of a modern LSTM


Forget Gate

Hidden state passed in from other cells

New input vector

Input Gate

Cell State

Output Gate

The math of the modern LSTM

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f)$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o)$$

$$h_t = o_t \tanh(c_t)$$


How many weights for a single LSTM unit?

\[ \text{number of weights} = 4|x| + 4|h| + 3 + 4 \]
How many weights for this network?

- Input: 50,000 word vocabulary, 4 LSTM layers of 100 cells per layer

- Compare that to a vanilla RNN with the same number of layers and vocabulary size....

- Can we shrink closer to a vanilla RNN but keep advantages of an LSTM?
GRU: A simplified LSTM

GRU: The Math

A linear interpolation between previous output and candidate output

$$h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t$$

Determines how much to make the output be influenced by the previous hidden state vs the current input.

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

Vector of all reset gates in the hidden layer

Determines how hard to reset this unit’s output

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

Vector of all outputs in the hidden layer

Candidate output

$$\tilde{h}_t = \tanh(W_{\tilde{h}} x_t + U_{\tilde{h}} (r_t \odot h_{t-1}))$$

LSTM/GRU Plusses and Minuses

- Lets networks handle problems with long term dependencies
- This lets LSTMs (or GRU) solve problems simple recurrent architectures cannot

- Still has trouble with XOR (time-delayed XOR where you XOR two inputs that are an unknown number of time steps apart)
- Lots of extra weights compared to regular cells
- Long and slow to train
- Not easy to inspect networks to understand them