Introduction to RNNs for NLP

SHANG GAO

About Me

- PhD student in the Data Science and Engineering program
- Took Deep Learning last year
- Work in the Biomedical Sciences, Engineering, and Computing group at ORNL
- Research interests revolve around deep learning for NLP
- Main project: information extraction from cancer pathology reports for NCI

Overview

- Super Quick Review of Neural Networks
- Recurrent Neural Networks
- Advanced RNN Architectures
  - Long-Short-Term-Memory
  - Gated Recurrent Units
- RNNs for Natural Language Processing
  - Word Embeddings
  - NLP Applications
- Attention Mechanisms and CNNs for Text

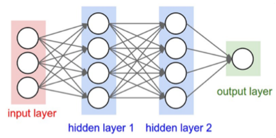
Neural Network Review

Neural networks are organized into **layers**

Each neuron receives signal from all neurons in the previous layer

Each signal connection has a weight associated with it based on how important it is; the more important the signal the higher the weight

These weights are the **model parameters**



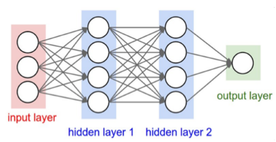
$$output = activation(\sum_{i}^{n}(input_i \times weight_i) + bias)$$

Neural Network Review

Each neuron gets the weighted sum of signals from the previous layer

The weighted sum is passed through the **activation function** to determine how much signal is passed to the next layer

The neurons at the very end determine the outcome or decision

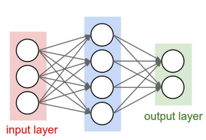


$$output = activation(\sum_{i}^{n}(input_i \times weight_i) + bias)$$

Feedforward Neural Networks

In a regular feedforward network, each neuron takes in inputs from the neurons in the previous layer, and then pass its output to the neurons in the next layer

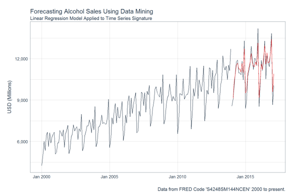
The neurons at the end make a classification based only on the data from the current input



What About Time Series Data?

In time series data, you have to consider patterns over time to effectively interpret the data:

- Weather data
- Stock market
- Speech audio
- Text and natural language
- Imaging and LIDAR for self-driving cars

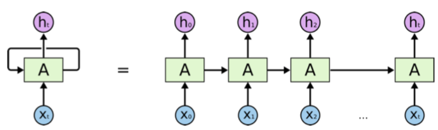


Recurrent Neural Networks

In a recurrent neural network, each neuron takes in data from the previous layer **AND** its own output from the previous timestep

The neurons at the end make a classification decision based on **NOT ONLY** the input at the current timestep **BUT ALSO** the input from all timesteps before it

Recurrent neural networks can thus capture patterns over time

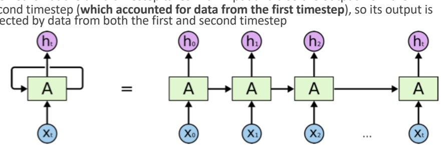


Recurrent Neural Networks

In the example below, the neuron at the first timestep takes in an input and generates an output

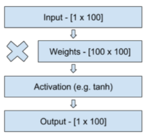
The neuron at the second timestep takes in an input **AND ALSO** the output from the first timestep to make its decision

The neuron at the third timestep takes in an input and also the output from the second timestep (**which accounted for data from the first timestep**), so its output is affected by data from both the first and second timestep

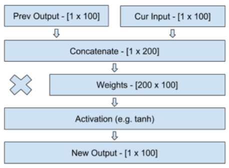


Recurrent Neural Networks

Feedforward:
output = sigmoid(weights * input + bias)



Recurrent:
output = sigmoid(weights * concat(input, previous_output) + bias)

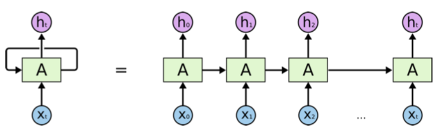


Recurrent Neural Networks

Another way to think of RNNs is just a very deep feedforward neural network, where each timestep adds another layer of depth.

- Every time there is another timestep, you concatenate the new input and then reapply the same set of weights

This is why with many timesteps, RNNs can become very slow to train.

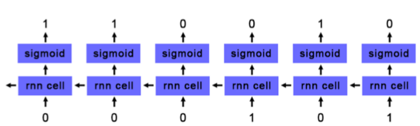


Toy RNN Example

Adding Binary

At each timestep, RNN takes in two values representing binary input

At each timestep, RNN outputs the sum of the two binary values taking into account any carryover from previous timestep

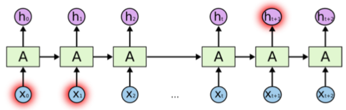


Problems with Basic RNNs

For illustrative purposes, let's assume at any given timestep, decision depends 50-50 on current input and previous output

RNN reads in input data (x_0) at the 1st timestep. The output (h_0) at the first timestep depends entirely on x_0

At the 2nd timestep, the output h_1 is influenced 50% by x_0 and 50% by x_1

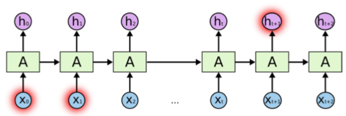


Problems with Basic RNNs

At the 3rd timestep, the output h_2 is influenced 25% by x_0, 25% by x_1, and 50% by x_2

The influence of x_0 decreases by half every additional timestep

By the end of the RNN, the data from the first timestep has very little impact on the output of the RNN



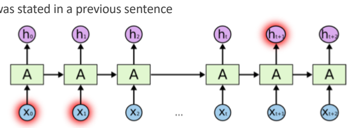
Problems with Basic RNNs

Basic RNN cells can't retain information across a large number of timesteps

In practice, RNNs can lose data in as few as 4-5 timesteps

This is causes problems on tasks where information needs to be retained over a long time

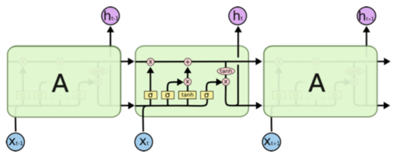
For example, in natural language processing, the meaning of a pronoun may depend on what was stated in a previous sentence



Long Short Term Memory

Long Short Term Memory cells are advanced RNN cells that address the problem of long-term dependencies

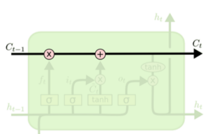
Instead of always writing to each cell at every time step, each unit has an internal 'memory' that can be written to selectively



Long Short Term Memory

Terminology:

- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t



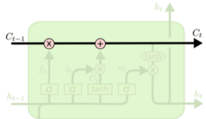
Long Short Term Memory

- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Input from the current timestep is written to the internal memory based on how relevant it is to the problem (relevance is learned during training through backpropagation)

If the input isn't relevant, no data is written into the cell

This way data can be preserved over many timesteps and be retrieved when it is needed

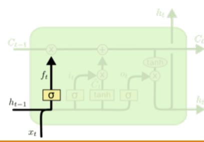


- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Long Short Term Memory

Movement of data into and out of an LSTM cell is controlled by “gates”

A gate is a sigmoid function that controls the flow of information through the LSTM
- Outputs a value between 0 (no flow) and 1 (let everything through)
- Each gate examines the input data and previous output to determine how information should flow through the LSTM



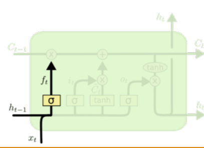
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Long Short Term Memory

The “forget gate” outputs a value between 0 (delete) and 1 (keep) and controls how much of the internal memory to keep from the previous timestep

For example, at the end of a sentence, when a ‘.’ is encountered, we may want to reset the internal memory of the cell



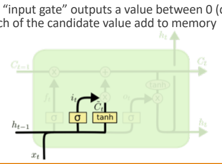
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Long Short Term Memory

The “candidate value” is the processed input value from the current timestep that may be added to memory
- Note that **tanh** activation is used for the “candidate value” to allow for negative values to subtract from memory

The “input gate” outputs a value between 0 (delete) and 1 (keep) and controls how much of the candidate value add to memory



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

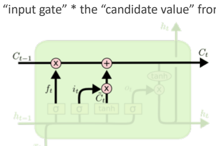
- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Long Short Term Memory

Combined, the "input gate" and "candidate value" determine what new data gets written into memory

The "forget gate" determines how much of the previous memory to retain

The new memory of the LSTM cell is the "forget gate" * the previous memory state + the "input gate" * the "candidate value" from the current timestep



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

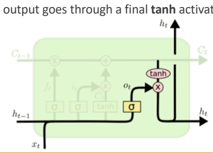
- x_t – input data at timestep t
- C_t – internal memory of LSTM at timestep t
- h_t – output of LSTM at timestep t

Long Short Term Memory

The LSTM cell does not output the contents of its memory to the next layer
- Stored data in memory might not be relevant for current timestep, e.g., a cell can store a pronoun reference and only output when the pronoun appears

Instead, an "output" gate outputs a value between 0 and 1 that determines how much of the memory to output

The output goes through a final **tanh** activation before being passed to the next layer



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

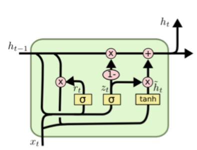
Gated Recurrent Units

Gated Recurrent Units are very similar to LSTMs but use two gates instead of three

The "update gate" determines how much of the previous memory to keep

The "reset gate" determines how to combine the new input with the previous memory

The entire internal memory is output without an additional activation



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTMs vs GRUs

Greff, et al. (2015) compared LSTMs and GRUs and found they perform about the same

Jozefowicz, et al. (2015) generated more than ten thousand variants of RNNs and determined that depending on the task, some may perform better than LSTMs

GRUs train slightly faster than LSTMs because they are less complex

Generally speaking, tuning hyperparameters (e.g. number of units, size of weights) will probably affect performance more than picking between GRU and LSTM

RNNs for Natural Language Processing

The natural input for a neural network is a vector of numeric values (e.g. pixel densities for imaging or audio frequency for speech recognition)

How do you feed language as input into a neural network?

The most basic solution is **one hot encoding**

```
                Paris
       Rome                      word V
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

One Hot Encoding LSTM Example

Trained LSTM to predict the next character given a sequence of characters

Training corpus: All books in Hitchhiker's Guide to the Galaxy series

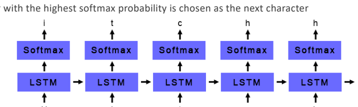
One-hot encoding used to convert each character into a vector

72 possible characters – lowercase letters, uppercase letters, numbers, and punctuation

Input vector is fed into a layer of 256 LSTM nodes

LSTM output fed into a softmax layer that predicts the following character

The character with the highest softmax probability is chosen as the next character



Generated Samples

700 iterations: ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae ae

4200 iterations: the sand and the said the sand and the said the sand and the said the sand and the said the sand and the said the

36000 iterations: seared to be a little was a small beach of the ship was a small beach of the ship was a small beach of the ship

100000 iterations: the second the stars is the stars to the stars in the stars that he had been so the ship had been so the ship had been

290000 iterations: started to run a computer to the computer to take a bit of a problem off the ship and the sun and the air was the sound

500000 iterations: "I think the Galaxy will be a lot of things that the second man who could not be continually and the sound of the stars

One Hot Encoding Shortcomings

One-hot encoding is lacking because it fails to capture semantic similarity between words, i.e., the inherent meaning of word

For example, the words "happy", "joyful", and "pleased" all have similar meanings, but under one-hot encoding they are three distinct and unrelated entities

What if we could capture the meaning of words within a numerical context?

Word Embeddings

Word embeddings are vector representations of words that attempt to capture semantic meaning

Each word is represented as a vector of numerical values

Each index in the vector represents some abstract "concept"
- These concepts are unlabeled and learned during training

Words that are similar will have similar vectors

	Masculinity	Royality	Youth	Intelligence
King	0.95	0.95	-0.1	0.6
Queen	-0.95	0.95	-0.1	0.6
Prince	0.8	0.8	0.7	0.4
Woman	-0.95	0.01	-0.1	0.2
Peasant	0.1	-0.95	0.1	-0.3
Doctor	0.12	0. 1	-0.2	0.95

Word2Vec

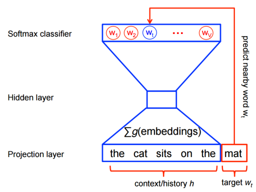
Words that appear in the same context are more likely to have the same meaning

- I am **excited** to see you today!
- I am **ecstatic** to see you today!

Word2Vec is an algorithm that uses a funnel-shaped single hidden layer neural network to create word embeddings

Given a word (in one-hot encoded format), it tries to predict the neighbors of that word (also in one-hot encoded format), or vice versa

Words that appear in the same context will have similar embeddings



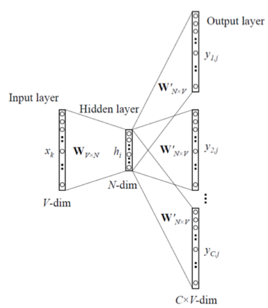
Word2Vec

The model is trained on a large corpus of text using regular backpropagation

For each word in the corpus, predict the 5 words to the left and right (or vice versa)

Once the model is trained, the embedding for a particular word is the row of the weight matrix associated with that word

Many pretrained vectors (e.g. Google) can be downloaded online



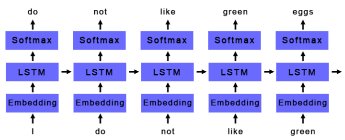
Word2Vec on 20 Newsgroups

Basic Deep Learning NLP Pipeline

Generate Word Embeddings
- Python gensim package

Feed word embeddings into LSTM or GRU layer

Feed output of LSTM or GRU layer into softmax classifier



Applications

Language Models
- Given a series of words, predict the next word
- Understand the inherent patterns in a given language
- Useful for autocompletion and machine translation

Sentiment Analysis
- Given a sentence or document, classify if it is positive or negative
- Useful for analyzing the success of a product launch or automated stock trading based off news

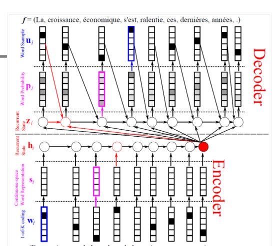
Other forms text classification
- Cancer pathology report classification

Advanced Applications

Question Answering
- Read a document and then answer questions
- Many models use RNNs as their foundation

Automated Image Captioning
- Given an image, automatically generate a caption
- Many models use both CNNs and RNNs

Machine Translation
- Automatically translate text from one language to another
- Many models (including Google Translate) use RNNs as their foundation

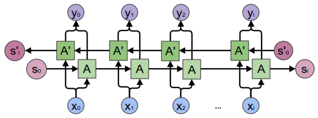


LSTM Improvements
Bi-directional LSTMs

Sometimes, important context for a word comes after the word (especially important translation)

- I saw a **crane** flying across the sky
- I saw a **crane** lifting a large boulder

Solution - use two LSTM layers, one that reads the input forward and one that reads the input backwards, and concatenate their outputs



LSTM Improvements
Attention Mechanisms

Sometimes only a few words in a sentence or document are important and the rest do not contribute as much meaning

- For example, when classifying cancer location from cancer pathology reports, we may only care about certain keywords like "right upper lung" or "ovarian"

In a traditional RNN, we usually take the output at the last timestep

By the last timestep, information from the important words may have been diluted, even with LSTMs and GRUs units

How can we capture the information at the most important words?

LSTM Improvements
Attention Mechanisms

Naïve solution: to prevent information loss, instead of using the LSTM output at the last timestep, take the LSTM output at every timestep and use the average

Better solution: find the important timesteps, and weight the output at those timesteps much higher when doing the average

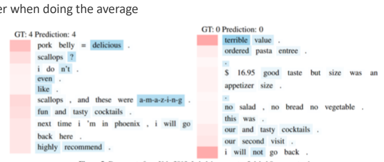
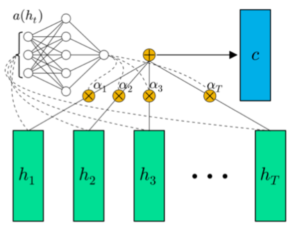


Figure 5: Documents from Yelp 2013. Label 4 means star 5, label 0 means star 1.

LSTM Improvements
Attention Mechanisms

An attention mechanism calculates how important the LSTM output at each timestep is

At each timestep, feed the output from the LSTM/GRU into the attention mechanism



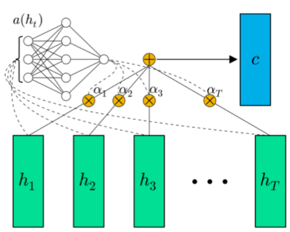
LSTM Improvements
Attention Mechanisms

There are many different implementations, but the basic idea is the same:

- Compare the input vector to some 'context' target vector
- The more similar the input is to the target vector, the more important it is
- For each input, output a single scalar value indicating it's importance

Common implementations:

- Additive: Single hidden layer neural network
- Dot product



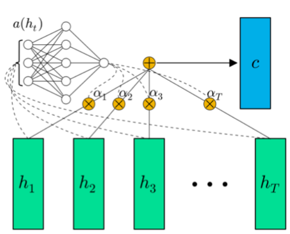
LSTM Improvements
Attention Mechanisms

Once we have the importance values from the attention mechanism, we apply softmax to normalize

- softmax always adds to 1

The softmax ouput tells us how to weight the output at each timestep, i.e., how important each timestep is

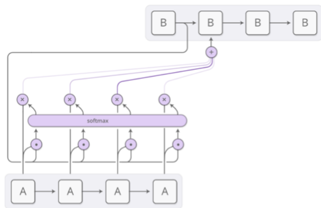
Multiply the output at each timestep with its corresponding softmax weight and add to create a weighted average



LSTM Improvements
Attention Mechanisms

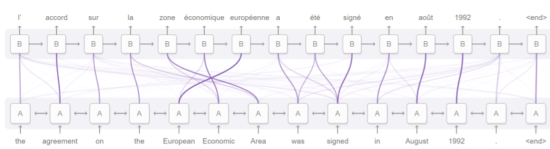
We initialize the "context" target vector based off the NLP application:

- For question answering, can represent a question being asked
- For machine translation, can represent the previous word or sentence
- For classification, can be initialized randomly and learned during training



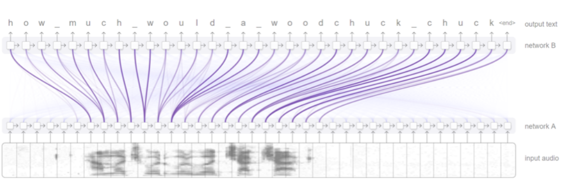
LSTM Improvements
Attention Mechanisms

With attention, you can visualize how important each timestep is for a particular task



LSTM Improvements
Attention Mechanisms

With attention, you can visualize how important each timestep is for a particular task

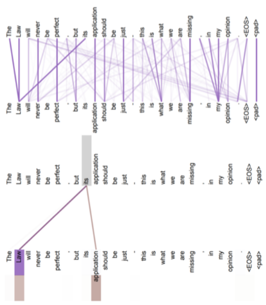


Self Attention

Self attention is a form of neural attention in which a sequence of words is compared against itself

This allows the network to learn important relationships between words in the same sequence, especially across long distances

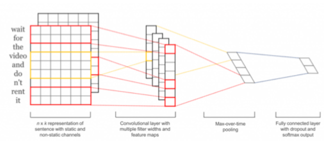
Self-attention is becoming popular in NLP because it can find long distance relationships like RNNs but is up to 10x faster to run.



CNNs for Text Classification

Start with Word Embeddings
- If you have 10 words, and your embedding size is 300, you'll have a 10x300 matrix

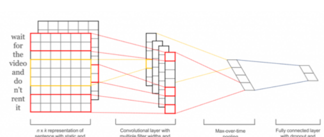
3 Parallel Convolution Layers
- Take in word embeddings
- Sliding window that processes 3, 4, and 5 words at a time (1D conv)
- Filter sizes are 3x300x100, 4x300x100, and 5x300x100 (width, in-channels, out-channels)
- Each conv layer outputs 10x100 matrix



CNNs for Text Classification

Maxpool and Concatenate
- For each filter channel, maxpool across the entire width of sentence
- This is like picking the 'most important' word in the sentence for each channel
- Also ensures every sentence, no matter how long, is represented by same length vector
- For each of the three 10x100 matrices, returns 1x100 matrix
- Concatenate the three 1x100 matrices into a 1x300 matrix

Dense and Softmax



Questions?

Cool Deep Learning Videos

Style Transfer - https://www.youtube.com/watch?v=Khui4ASldmU

4 experiments where AI outsmarted its creators - https://www.youtube.com/watch?v=GdTBqBnqhaQ

One Pixel Attack - https://www.youtube.com/watch?v=SA4YEAWVpbk