CSC321 Neural Networks and Machine Learning

Lecture 10

March 18, 2020

Agenda

- GloVe: Word Embeddings Revisisted
- Recurrent Neural Networks for Classification
- Gradient Explosion and Vanishing
- Long Short-Term Memory

Course Continuity FAQ

1. What will happen to lectures, tutorials, and office hours?

They will all be held online on Quercus. They will be recorded whenever possible.

2. What will happen to the final exam?

We're still working with the department to figure out what to do. For now, you should study as if there will be a final exam.

3. I need accommodations due to recent events or due to the course being moved online.

Please email Lisa, so we can try our best to help you learn machine learning.

I need help

Please reach out to Lisa if you need help! I'm happy to discuss individually (online) about your situation, or about the course material if you can't make office hours.

Emergency Assistance Grants (just announced yesterday):

https://future.utoronto.ca/finances/financial-aid/emergency-assistance-grants/

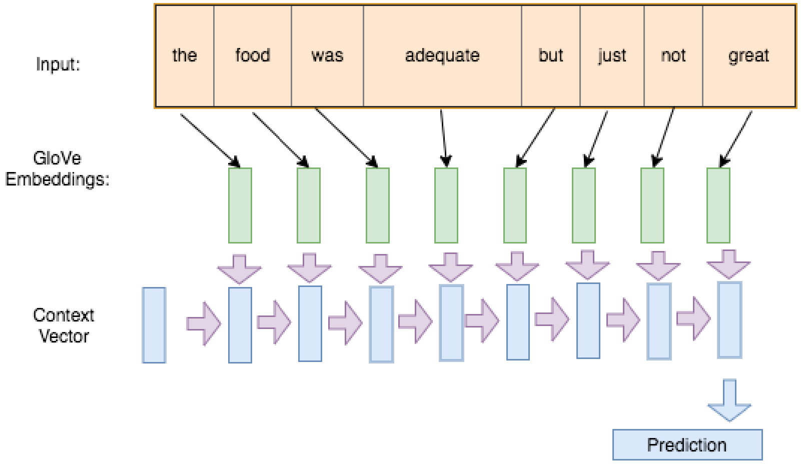
Other Updates

- Remark requests are delayed
- Homework 4 will be graded early next week
- Project 3 is due soon
- Homework 5 released
- Project 4 will be released soon

Word Embeddings Revisisted (GloVe)

Why use word embeddings?

Recall this example of a discriminative RNN from the last class:



How should we represent each input $\mathbf{x}^{(t)}$?

Representing Inputs

Idea 1: Use one-hot encodings to represent each $x^{(t)}$

If we were representing a sentence using a sequence of **characters**, then one-hot encodings could work.

However, there are tens of thousands of words in the English language!

Why is this a problem? (Hint: overfitting)

Using Word Embeddings

Idea 2: Learn *word embeddings*, or low-dimensional vector representations of words.

We could do this in two ways:

- Learn the word embeddings *at the same time* as our recurrent neural network
  - Forward pass includes word embedding lookup, RNN computation, prediction computation, and loss computation
  - Compute the derivatives of the loss with respect to the word embeddings, as in any other network
- Use pre-trained word embeddings
  - Like the ones we trained in Project 2!

GloVe Embeddings

GloVe: Global Vectors for Word Representation

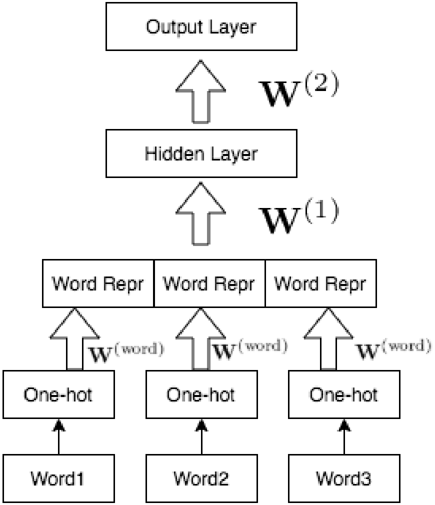
https://nlp.stanford.edu/projects/glove/

Most commonly used pre-trained word embeddings nowadays

The training of GloVe is a bit different from how we trained our word embeddings in P2

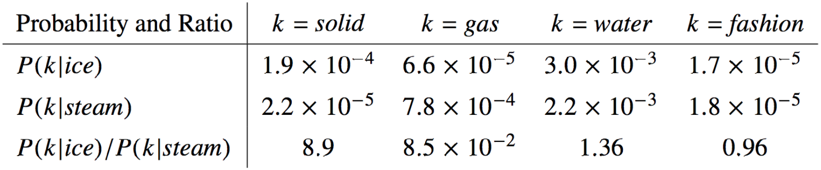
Project 2 Word Embeddings

Low-dimensional word embedding trained on the task of predicting the **next word** given the previous 3 words.



GloVe Embedding Training

Key insight: The co-occurrence matrix of words contain information about the semantic information (meaning) of words



Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

In particular, the *ratio* of co-occurrences encodes semantic information!

So, train word embeddings to be able to predict word co-occurrences. (Details are omitted here, but you can read the paper here https://nlp.stanford.edu/projects/glove/)

GloVe Embedding Demo

Demo on Google Colab

Key idea from the Demo

- Distances are somewhat meaningful, and are based on **word co-occurences**
  - the words “black” and “white” will have similar embeddings because they co-occur with similar other words.
  - “cat” and “dog” is more similar to each other than “cat” and “kitten” because the latter two words occur in *different contexts*!
- Word Analogies: Directions in the embedding space can be meaningful
  - “king” - “man” $+$ “woman” $\approx$ “queen”
- Bias in Word Embeddings (and Neural Networks in General)
  - neural networks pick up pattern in the data
  - these patterns can be biased and discriminatory

Bias and Fairness

Word embeddings are inherently biased because there is bias in the training data.

Neural networks learn patterns in the training data, so if the training data contains human biases, then so will the trained model! This effect was seen in:

- criminal sentencing (certain demographics are treated more harshly by the trained model model)
- resume filtering
- facial recognition technology

Fairness

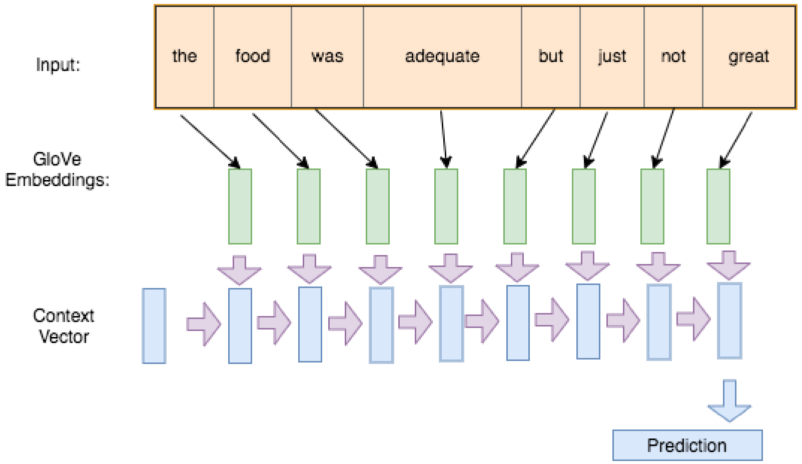
Suppose we want to deploy our model from Project 3 to help people who are visually impaired.

How might our model from project 3 be **unfair**?

Sentiment Analysis with Recurrent Neural Networks

Sequence Classification

Let's build this model:



Sentiment140 Data

Dataset of tweets with either a positive or negative emoticon, but with the emoticon removed.

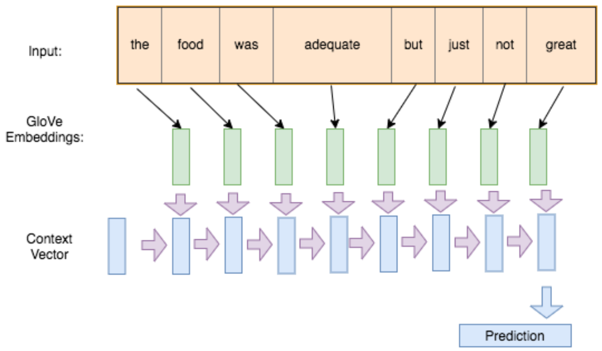
Input: Tweet (sequence of words/characters)

Target: Positive or negative emoticon?

Example:

- Negative: “Just going to cry myself to sleep after watching Marley and Me”
- Positive: “WOOOOO! Xbox is back”

Approach



- Use GloVe embeddings to represent words as input $\mathbf{x}^{(t)}$ (note: we could have chosen to work at the character level)
- Use a recurrent neural network to get a combined embedding of the *entire* tweet
- Use a fully-connected layer to make predictions (happy vs sad)

Key Considerations

- We'll be using the PyTorch `nn.RNN` module, which can be unintuitive
- Batching difficulties: each tweet is a different length, so how can we batch?
    - One way is to *pad* shorter sequences with a special "padding" token at the end of the sequence
    - However, we want to minimize this padding due to computational complexity
    - We'll do something simple today, and use PyTorch `torchtext` later

Demo

Key Takeaways

You should be able to understand. . .

- why we want to use RNNs rather than CNN/MLP
- why/how GloVe embeddings are used in RNNs
- what the hidden state computations depend on (not the exact computation, but the dependencies)
- which weights are shared and which weights are not
- why batching is trickier when training an RNN (compared to training a CNN/MLP)

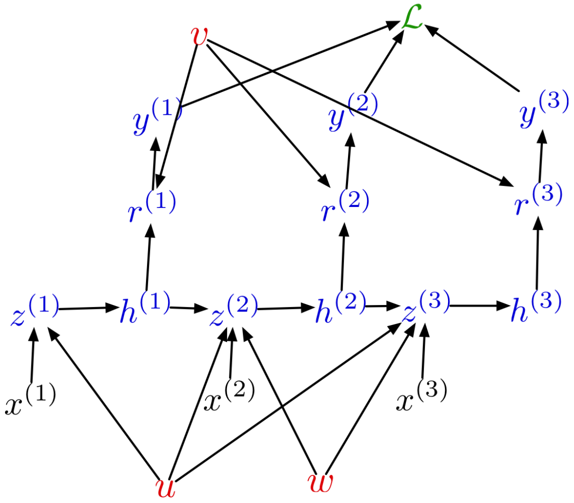
You don't need to. . .

- memorize the code you saw, or the PyTorch API
- know the exact math behind RNNs

Gradient Explosion and Vanishing

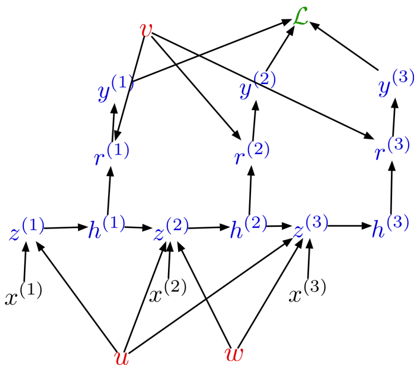
RNN Gradients

Last time we saw the unrolled computation graph for a small RNN:



$$\begin{tikzcd}
 & & v \arrow[ddl] \arrow[ddr] \arrow[ddrrr] & & \mathcal{L} & \\
 & y^{(1)} \arrow[urrr] & & y^{(2)} \arrow[ur] & & y^{(3)} \arrow[ul] \\
 & r^{(1)} \arrow[u] & & r^{(2)} \arrow[u] & & r^{(3)} \arrow[u] \\
z^{(1)} \arrow[r] & h^{(1)} \arrow[r] \arrow[u] & z^{(2)} \arrow[r] & h^{(2)} \arrow[r] \arrow[u] & z^{(3)} \arrow[r] & h^{(3)} \arrow[u] \\
x^{(1)} \arrow[u] & & x^{(2)} \arrow[u] & & x^{(3)} \arrow[u] & \\
 & u \arrow[uul] \arrow[uur] \arrow[uurrr] & & w \arrow[uul] \arrow[uur] & &
\end{tikzcd}$$

Backprop Through Time



Activations:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y^{(t)}} = \overline{\mathcal{L}}\,\frac{\partial \mathcal{L}}{\partial y^{(t)}}$$

$$\overline{r^{(t)}} = \overline{y^{(t)}}\,\phi'(r^{(t)})$$

$$\overline{h^{(t)}} = \overline{r^{(t)}}\,v + \overline{z^{(t+1)}}\,w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\,\phi'(z^{(t)})$$

Parameters:

$$\overline{u} = \sum_t \overline{z^{(t)}}\,x^{(t)}$$

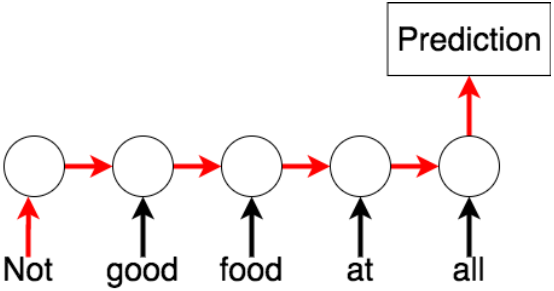
$$\overline{v} = \sum_t \overline{r^{(t)}}\,h^{(t)}$$

$$\overline{w} = \sum_t \overline{z^{(t+1)}}\,h^{(t)}$$

Key idea: multivariate chain rule!

Gradient Explosion and Vanishing

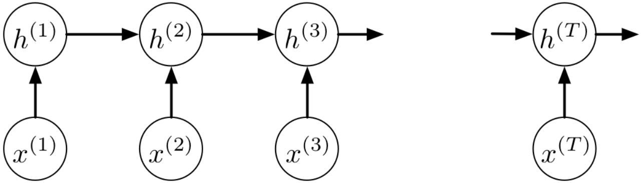
The longer your sequence, the longer gap the time step between when we see potentially important information and when we need it:



The derivatives need to travel this entire pathway.

Why Gradients Explode or Vanish

Consider a univariate version of the RNN:



Backprop updates:

$$\overline{h^{(t)}} = \overline{z^{(t+1)}}\, w$$
$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Applying this recursively:

$$\overline{h^{(1)}} = w^{T-1} \phi'(z^{(2)}) \cdots \phi'(z^{(T)}) \overline{h^{(T)}}$$

With linear activations:

$$\frac{\partial h^{(T)}}{\partial h^1} = w^{T-1}$$

Exploding:

$$w = 1.1,\ T = 50 \Rightarrow \frac{\partial h^{(T)}}{\partial h^{(1)}} = 117.4$$

Vanishing:

$$w = 0.9,\ T = 50 \Rightarrow \frac{\partial h^{(T)}}{\partial h^{(1)}} = 0.00515$$

Multivariate Hidden States

More generally, in the multivariate case, the **Jacobians** multiply:

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}}$$

Matrices can "explode" or "vanish" just like scalar values, though it's slightly harder to make precise.

(For this course, you don't need to know what Jacobians are; just that the idea from the previous slide applies to the case where **h** is a vector.)

Repeated Application of Functions

Another way to look at why gradients explode or vanish is that we are applying a function over and over again.

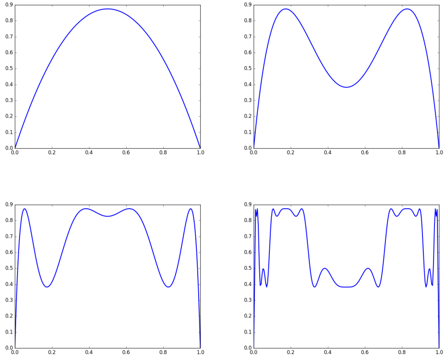
Each hidden layer computes some function of previous hidden layer and the current input: $\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$

This function gets repeatedly applied:

$$\begin{aligned}
\mathbf{h}^{(4)} &= f(\mathbf{h}^{(3)}, \mathbf{x}^{(4)}) \\
&= f(f(\mathbf{h}^{(2)}, \mathbf{x}^{(3)}), \mathbf{x}^{(4)}) \\
&= f(f(f(\mathbf{h}^{(1)}, \mathbf{x}^{(2)}), \mathbf{x}^{(3)}), \mathbf{x}^{(4)})
\end{aligned}$$

Iterated Functions

We get complicated behaviour from iterated functions. Consider $f(x) = 3.5x(1-x)$



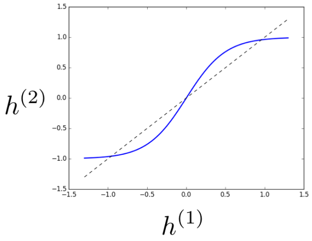
Note that the function values gravitate towards **fixed points**, and that the derivatives becomes either **very large** or **very small**.

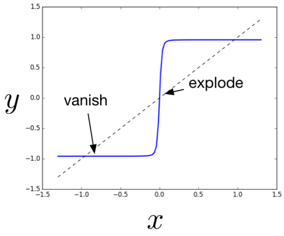
RNN with tanh activation

More concretely, consider an RNN with a tanh activation function:

$$\begin{tikzcd}
 & & & & y \\
h^{(1)} \arrow[r, "2"] & h^{(2)} \arrow[r, "2"] & h^{(3)} \arrow[r, "2"] & h^{(4)} \arrow[r, "2"] & h^{(5)} \arrow[u, "1"'] \\
x \arrow[u, "1"] & & & &
\end{tikzcd}$$

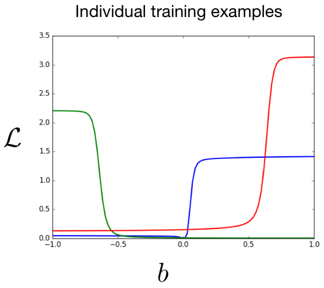
The function computed by the network:

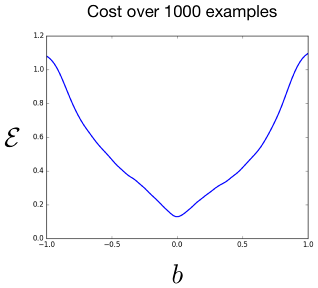




Cliffs

Repeatedly applying a function adds a new type possible loss landscape: **cliffs**, where the gradient of the loss with respect to a parameter is either close to 0, or very large.



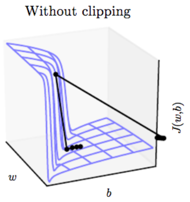


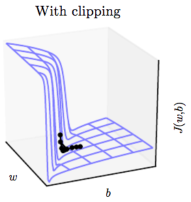
Generally, the gradient will explode on some inputs and vanish on others. In expectation, the cost may be fairly smooth.

Gradient Clipping

One solution is to "clip" the gradient so that it has a norm of at most η. Otherwise, update the gradient $\mathbf{g}$ with $\mathbf{g} \leftarrow \eta \frac{\mathbf{g}}{||\mathbf{g}||}$

The gradients are biased, but at least they don't blow up:





Gradient clipping solves the exploding gradient problem, but not the vanishing gradient problem.

Learning Long-Term Dependencies

Idea: Initialization

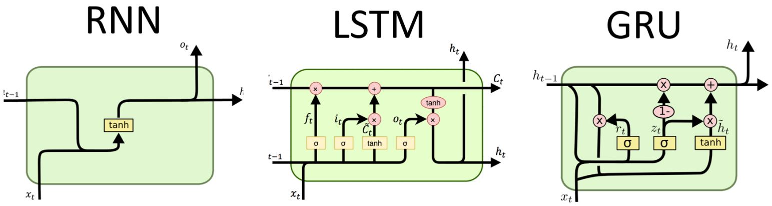
Hidden units are a kind of memory. Their default behaviour should be to **keep their previous value**.

If the function $\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$ is close to the identity, then the gradient computations $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}}$ are stable.

This initialization allows learning much longer-term dependencies than “vanilla” RNNs

Long-Term Short Term Memory

Change the **architectrue** of the recurrent neural network by replacing each single unit in an RNN by a “memory block”:



Two such blocks:

- LSTM: Long-Term Short Term Memory
- GRU: Gated Recurrent Units

Gating

The key idea behind these memory blocks is **gating**:

- A forget/write gate that computes how much information to forget/write
- Use of sigmoid activations to obtain values between 0 and 1

Visualization:
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Demo (with GRU/LSTM)

Key Takeaways

You should be able to understand...

- why learning long-term dependencies is hard in a vanilla RNN
- why gradients vanish/explode in a vanilla RNN
- what cliffs are and how repeated application of a function generates cliffs
- what gradient clipping is and when it is useful
- the idea behind gating

You do not need to know...

- the exact formula for how to update a vanilla RNN, LSTM, or GRU hidden state
- what Jacobians are or how to compute them