

CSC321 Neural Networks and Machine Learning

Lecture 9

March 11, 2020

Agenda

- ▶ Next few tutorials
- ▶ Review autoencoder
- ▶ Optimizing the input
 - ▶ Guided Backprop
 - ▶ Adversarial examples
- ▶ Recurrent Neural Networks

Tutorials

- ▶ **Tutorial 9:** Hands-on lab on transfer learning
- ▶ **Tutorial 10:** Cumulative Review of weeks 4-9 materials.
Prizes!
- ▶ **Tutorial 11:** Hands-on lab on recurrent neural networks.
Requests?
- ▶ **Tutorial 12:** Review for the final exam.

Exam Jam: April 3rd, 5pm-6pm in DH2060

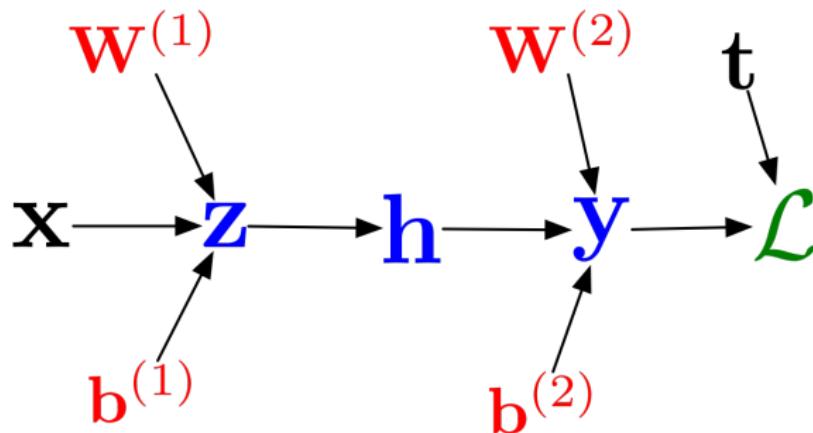
Review: Autoencoder

- ▶ What is the purpose of an autoencoder?
- ▶ What is the purpose of transpose convolutions in an autoencoder?
- ▶ What happens if an autoencoder overfits?
- ▶ Why do autoencoders produce blurry images?

Optimizing the Input

Computation Graph

Recall the computation graph of a two-layer MLP:

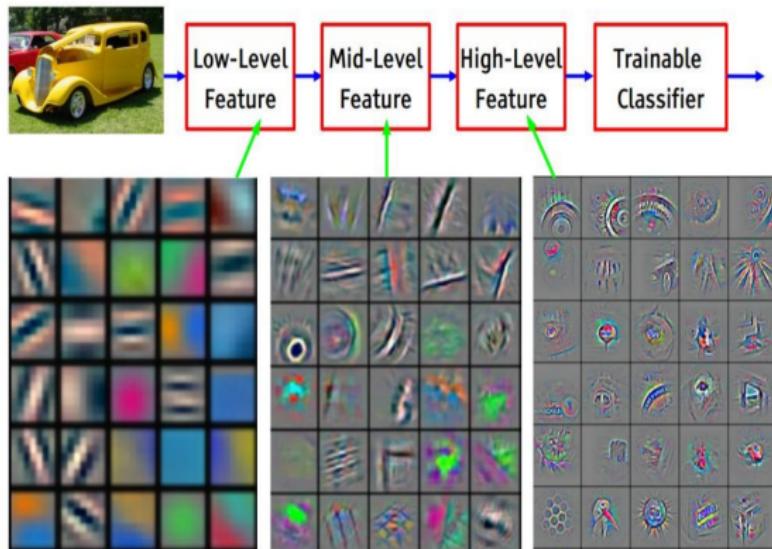


Suppose the weights have already been trained.

From this graph, you could compute $\bar{x} = \frac{\partial \mathcal{L}}{\partial x}$, but we never made use of this.

First hour: lots of fun things you can do by running gradient descent/ascent on the input!

Review: Feature Visualization



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

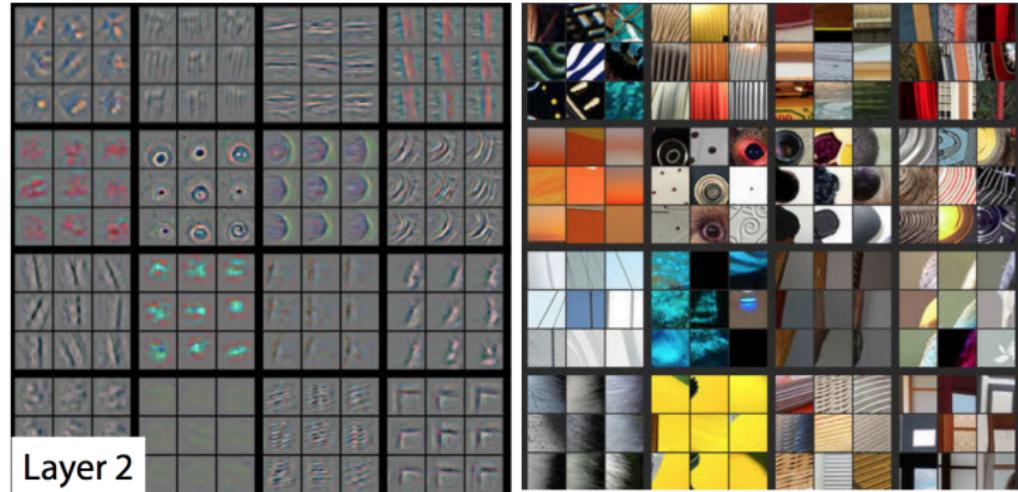
Review: Feature Visualization



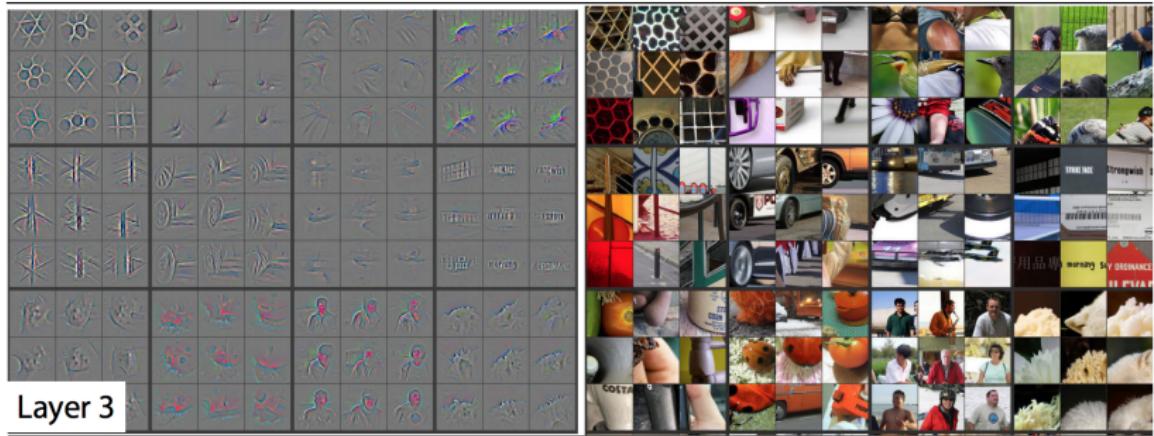
Layer 1



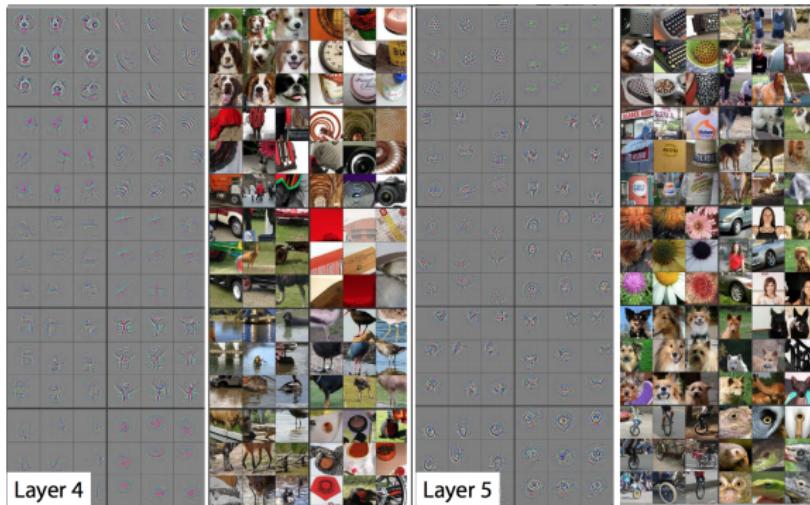
Review: Feature Visualization



Review: Feature Visualization



Review: Feature Visualization



Feature Visualization

- ▶ Higher layers seem to pick up more abstract, high-level information.
- ▶ Problem with looking at the **region** of the image that leads to a high unit activation:
 - ▶ Can't tell what the unit is actually responding to in the image
 - ▶ We may read too much into the results, e.g. a unit may detect red, and the images that maximize its activation will all be stop signs.
- ▶ Can use input gradients to diagnose what the unit is responding to.

Image Gradient (for image generation?)

Can we generate images by:

1. Start with a random image
2. Use gradient descent to tune the random image, so a neural network predicts that the image is of class (e.g. Samoyed)

Let's see if this idea works: Demo (see posted html notes)

Image Gradient (for image generation?)

Can we generate images by:

1. Start with a random image
2. Use gradient descent to tune the random image, so a neural network predicts that the image is of class (e.g. Samoyed)

Let's see if this idea works: Demo (see posted html notes)

Nope! Classification is a much easier task than generation. These gradients aren't very meaningful.

Image Gradient (to see which pixels would affect an activation)

Can we learn about the input gradients of a (natural) image? (That is, how can we make a Samoyed dog “more like a Samoyed dog”?)

- ▶ Take a network like AlexNet
- ▶ Compute the gradient of $\log p(y = \text{samoyed} | \mathbf{x})$
- ▶ Visualize the gradient

Demo (see posted html notes)

Image Gradient (to see which pixels would affect an activation)

Can we learn about the input gradients of a (natural) image? (That is, how can we make a Samoyed dog “more like a Samoyed dog”?)

- ▶ Take a network like AlexNet
- ▶ Compute the gradient of $\log p(y = \text{samoyed} | \mathbf{x})$
- ▶ Visualize the gradient

Demo (see posted html notes)

The image gives some idea of where AlexNet detects “Samoyed”, but is not that meaningful.

Why gradients are difficult to interpret

One explanation is that network tries to detect samoyeds everywhere.

A pixel may be consistent with samoyed in one location (positive gradient), but inconsistent with samoyed in other locations (negative gradient).

The positive and negative gradients cancel out.

The full explanation is beyond the scope of this course.

Guided Backprop

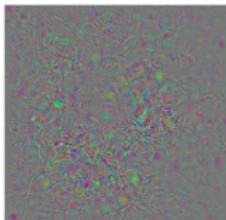
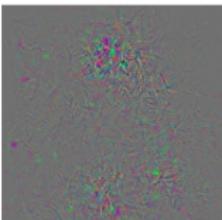
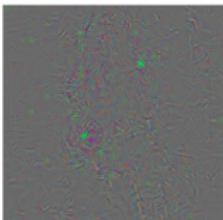
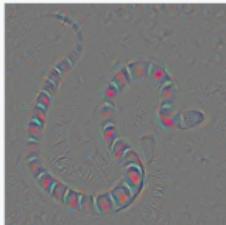
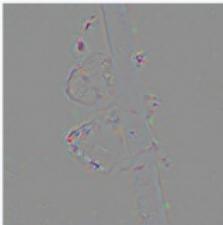
Guided backprop is a total hack to prevent this cancellation.

Idea: Compute the backward pass, but **only allow positive gradient signals to pass through**. Set all negative gradient signals to zero.

We want to visualize what **excites** a given unit, not what **suppresses** it!

Note: this isn't really the gradient of anything!

Guided Backprop Examples

	Target class: King Snake (56)	Target class: Mastiff (243)	Target class: Spider (72)
Original Image			
Colored Vanilla Backpropagation			
Colored Guided Backpropagation (GB)			

Other Ideas

Interpreting CNN's is a difficult task, but there are many ideas that people have tried:

<https://github.com/utkuozbulak/pytorch-cnn-visualizations>

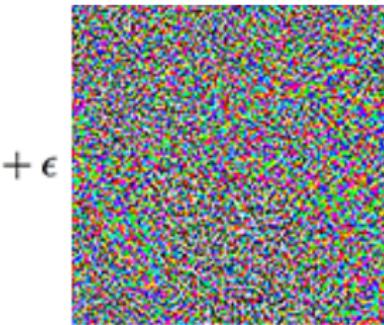
Adversarial Examples

What are these images of?



“panda”

57.7% confidence



=



“gibbon”

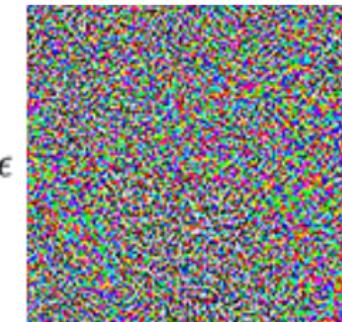
99.3% confidence

What are these images of?



“panda”

57.7% confidence



$+\epsilon$



“gibbon”

99.3% confidence

Producing adversarial images: Given an image for one category (e.g. panda), compute the image gradient to maximize the network's output unit for a different category (e.g. gibbon)

Non-targetted Adversarial Attack

Goal: Choose a small perturbation ϵ on an image x so that a neural network f misclassifies $x + \epsilon$.

Approach:

Use the same optimization process to choose ϵ to **minimize** the probability that

$$f(x + \epsilon) = \text{correctclass}$$

Targeted Adversarial Attack

Targeted attack

Maximize the probability that $f(x + \epsilon) = \text{target}$ incorrect class

Non-targeted attack

Minimize the probability that $f(x + \epsilon) = \text{correct}$ class

Demo (see html notes)

Adversarial Attack

- ▶ 2013: ha ha, how cute!
 - ▶ The paper which introduced adversarial examples was titled "Intriguing Properties of Neural Networks."
- ▶ 2018+: serious security threat
 - ▶ Nobody has found a reliable method yet to defend against them!
 - ▶ 7 of 8 proposed defenses accepted to ICLR 2018 were cracked within days.

White-box vs Black-box Adversarial Attacks

Adversarial examples transfer to different networks trained on a totally separate training set!

White-box Adversarial Attack: Model architecture and weights are known, so we can compute gradients. (What we've been doing so far in the demos)

Black-box Adversarial Attack: Model architecture and weights are unknown.

- ▶ You don't need access to the original network!
- ▶ You can train up a new network to match its predictions, and then construct adversarial examples for that.

Attack carried out against proprietary classification networks accessed using prediction APIs (MetaMind, Amazon, Google)

Adversarial Examples in 3D

It is possible to have a 3D object that gets misclassified by a neural network from all angles.

https://www.youtube.com/watch?v=piYnd_wYIT8

Printed Adversarial Examples

It is possible for a printed image to cause object detection to fail.

<https://www.youtube.com/watch?v=MIbFvK2S9g8>

Recurrent Neural Networks

Goal and Overview

Sometimes we're interested in making predictions about data in the form of **sequences**.

Examples:

- ▶ Given the price of a stock in the last week, predict whether stock price will go up
- ▶ Given a sentence (sequence of chars/words) predict its sentiment
- ▶ Given a sentence in English, translate it to French

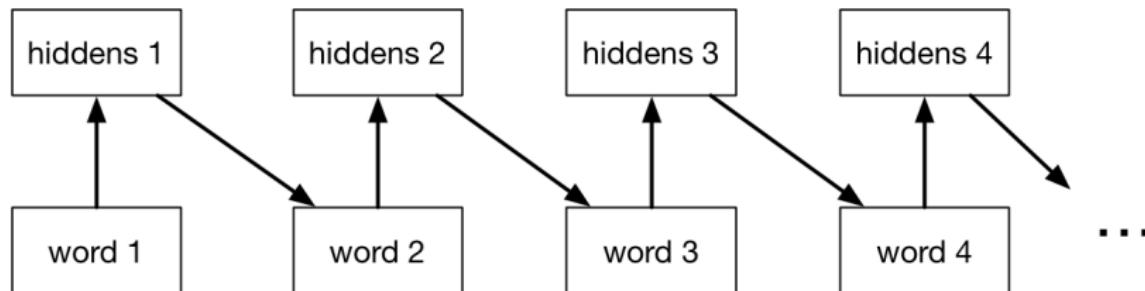
This last example is a **sequence-to-sequence prediction** task, because both inputs and outputs are sequences.

Language Model

We have already seen neural language models that make the **Markov Assumption**

$$p(w_i | w_1, \dots, w_{i-1}) = p(w_i | w_{i-3}, w_{i-2}, w_{i-1})$$

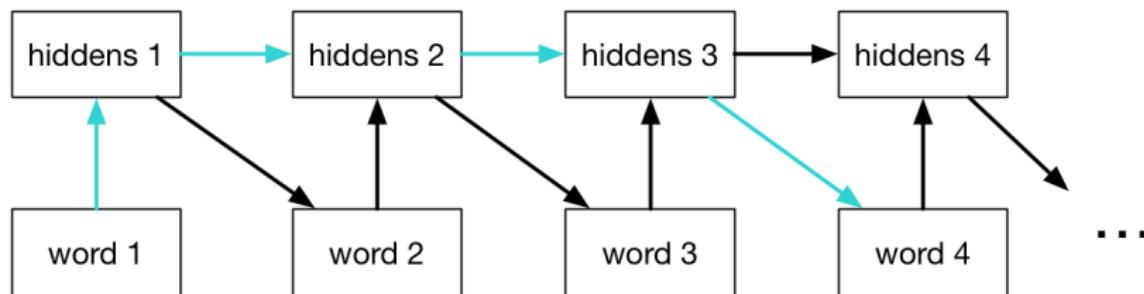
This means the model is **memoryless**, so they can only use information from their immediate context (in this figure, context length = 1):



Recurrent Neural Network

But sometimes long-distance context can be important.

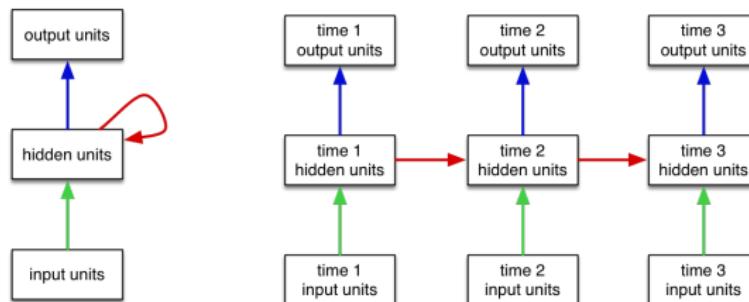
If we add connections between the hidden units, it becomes a **recurrent neural network (RNN)**. Having a memory lets an RNN use longer-term dependencies:



RNN Diagram

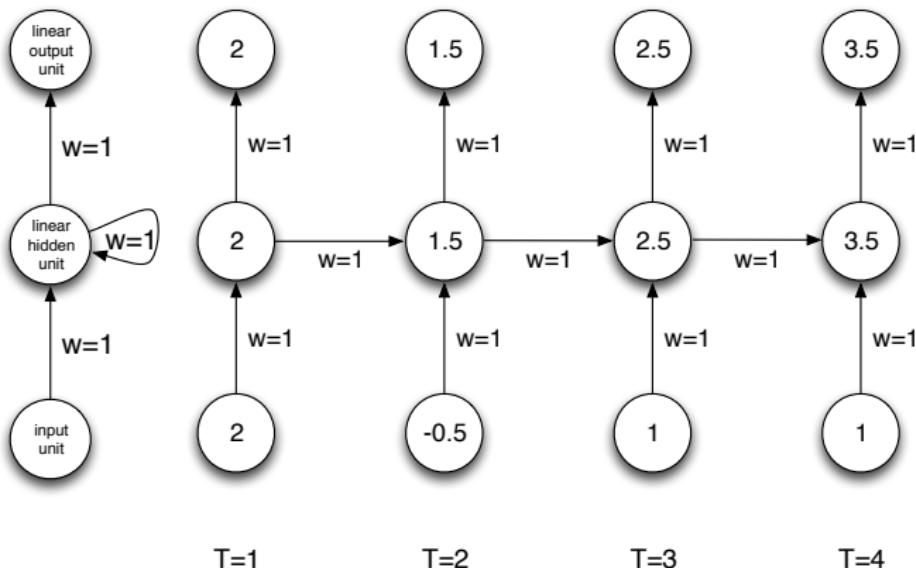
We can think of an RNN as a dynamical system with one set of hidden units which feed into themselves. The network's graph would then have self-loops.

We can **unroll** the RNN's graph by explicitly representing the units at all time steps. The weights and biases are shared between all time steps



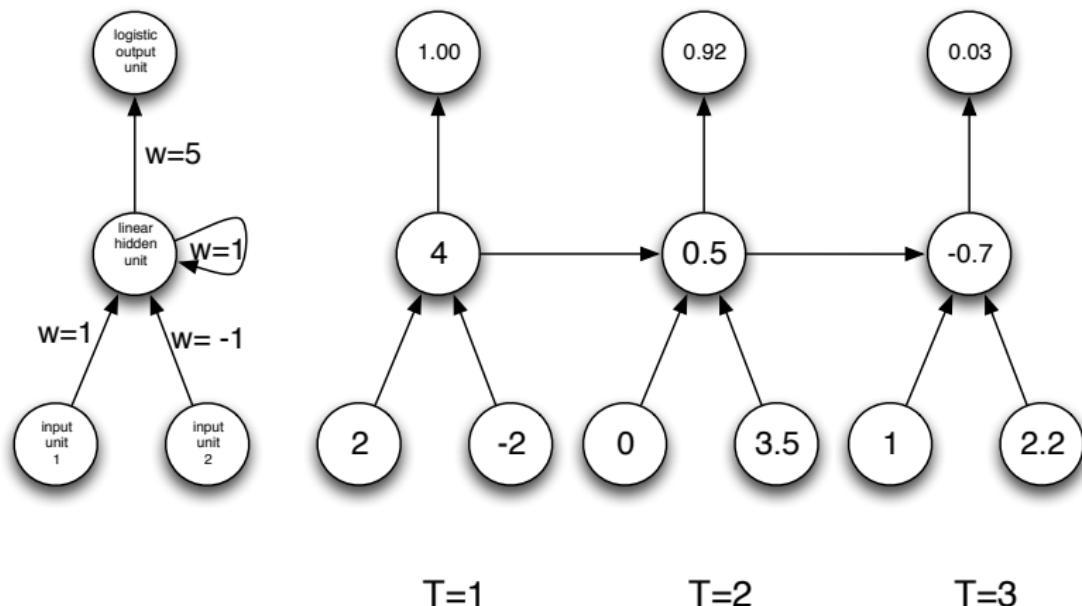
Simple RNN Example: Sum

This simple RNN takes a sequence of numbers as input (scalars), and sums its inputs.



Simple RNN Example 2: Comparison

This RNN takes a sequence of **pairs of numbers** as input, and determines if the total values of the first or second input are larger:



Simple RNN Example 3: Parity

Assume we have a sequence of binary inputs. We'll consider how to determine the **parity**, i.e. whether the number of 1's is even or odd.

We can compute parity incrementally by keeping track of the parity of the input so far:

Parity bits: 0 1 1 0 1 1 →
Input: 0 1 0 1 1 0 1 0 1 1

Each parity bit is the XOR of the input and the previous parity bit.

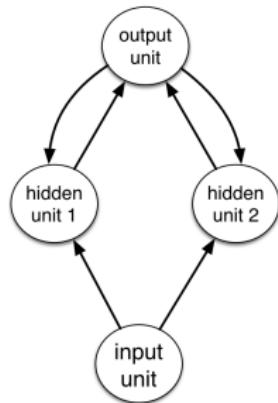
Parity is a classic example of a problem that's hard to solve with a shallow feed-forward net, but easy to solve with an RNN.

Parity Approach

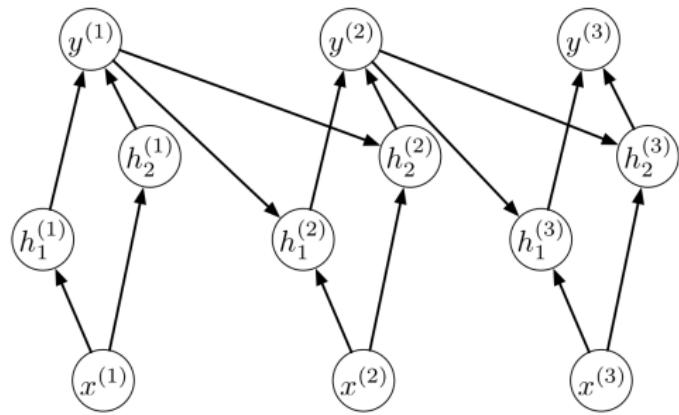
Let's find weights and biases for the RNN on the right so that it computes the parity. All hidden and output units are **binary threshold units** ($h(x) = 1$ if $x > 0$ and $h(x) = 0$ otherwise).

Strategy

- ▶ The output unit tracks the current parity, which is the XOR of the current input and previous output.
- ▶ The hidden units help us compute the XOR.



Unrolling Parity RNN



Parity Computation

The output unit should compute the XOR of the current input and previous output:

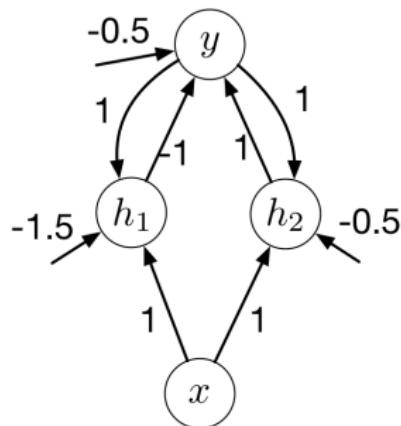
$y^{(t-1)}$	$x^{(t)}$	$y^{(t)}$
0	0	0
0	1	1
1	0	1
1	1	0

Computing Parity

Let's use hidden units to help us compute XOR.

- ▶ Have one unit compute AND, and the other one compute OR.
 - ▶ Then we can pick weights and biases just like we did for multilayer perceptrons.

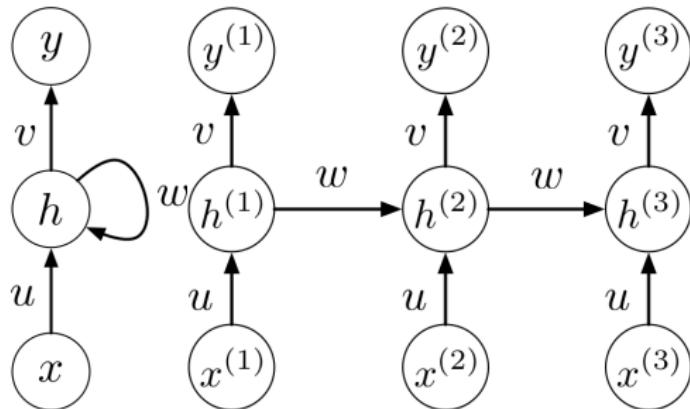
$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



Back Propagation Through Time

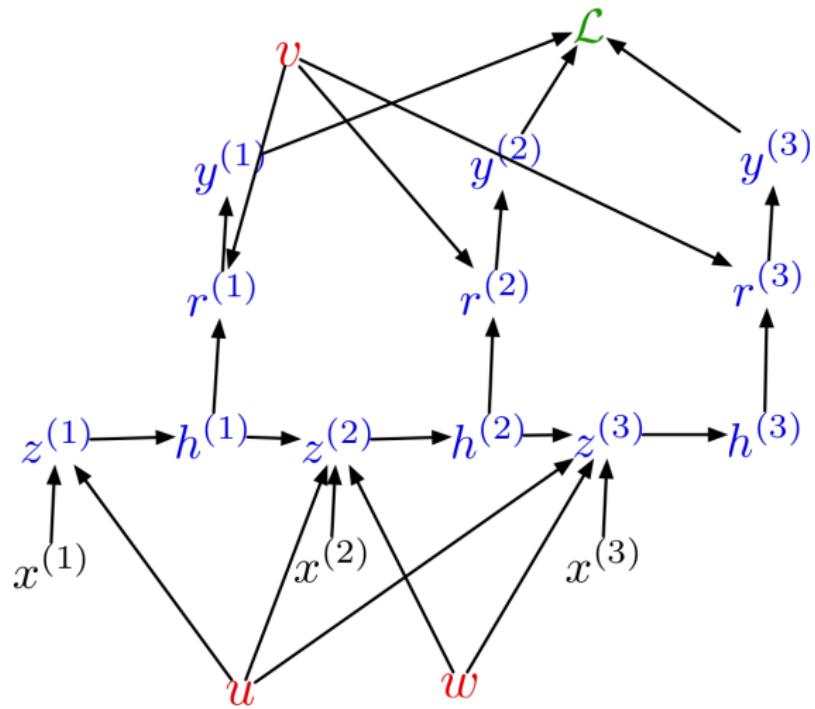
As you can guess, we don't usually set RNN weights by hand. Instead, we learn them using backprop.

In particular, we do backprop on the unrolled network. This is known as **backprop through time**.



Unrolled BPTT

Here's the unrolled computation graph. Notice the weight sharing.



RNN for language modelling

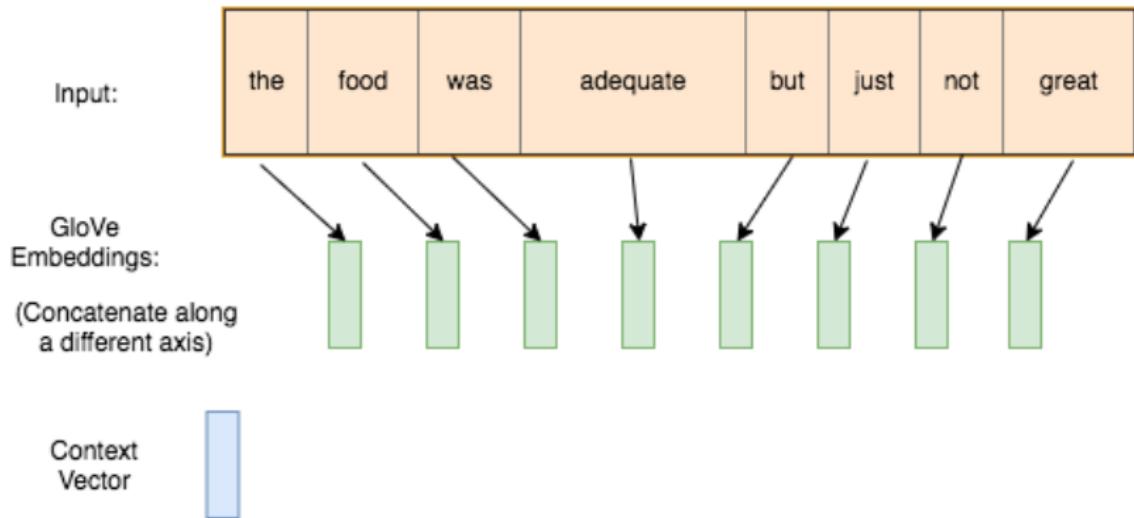
Usually, the sequence of inputs x_t will be **vectors**. The hidden states h_t are also vectors.

For example, we might use a sequence of one-hot vectors \mathbf{x}_t of words (or characters) to represent a sentence.

How would we use a RNN to determine (say) the sentiment conveyed by the sentence?

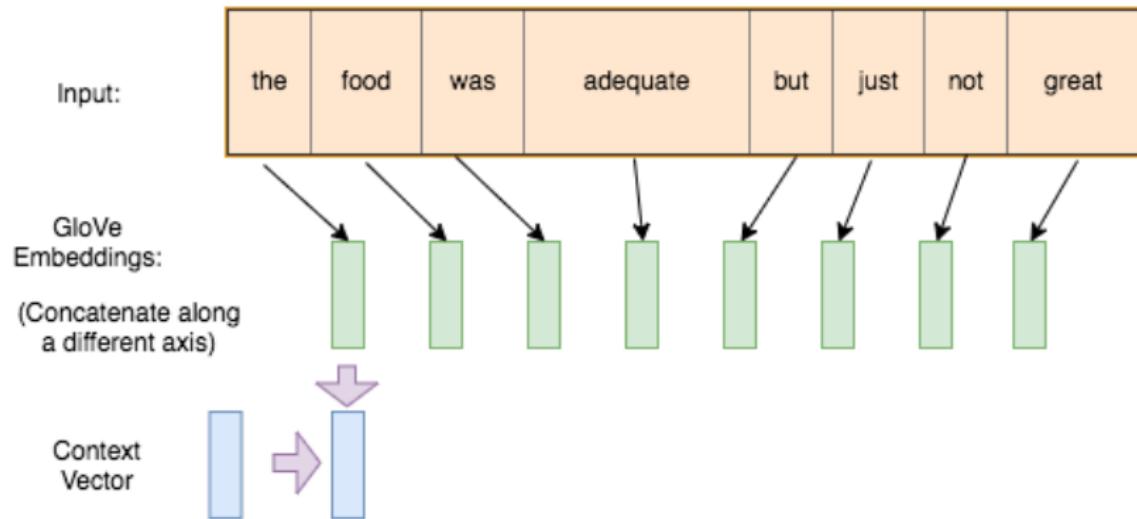
As usual, start with the forward pass...

RNN: Initial Hidden State



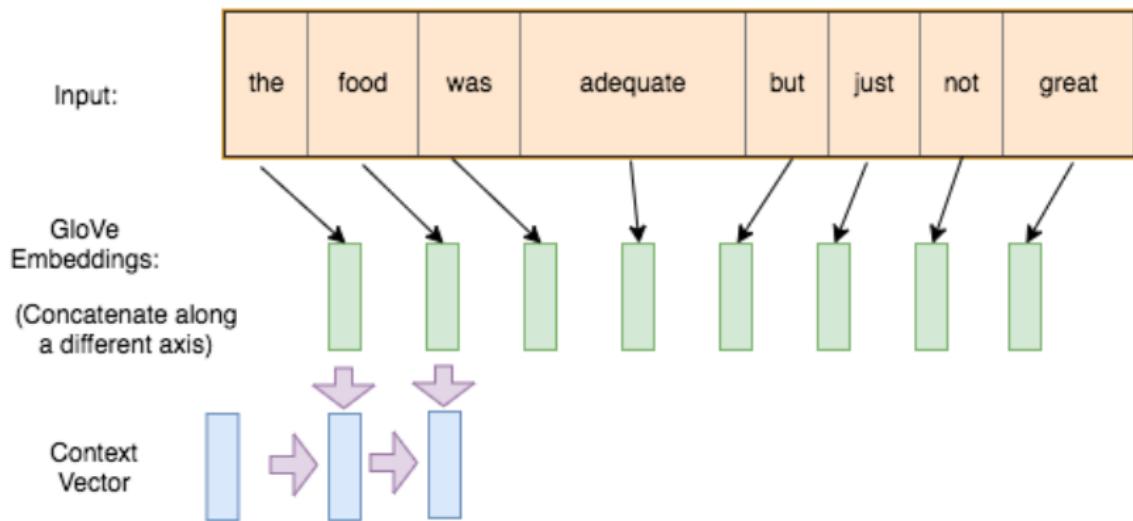
Start with an initial **hidden state** with a blank slate (can be a vector of all zeros, or a parameter that we train)

RNN: Update Hidden State



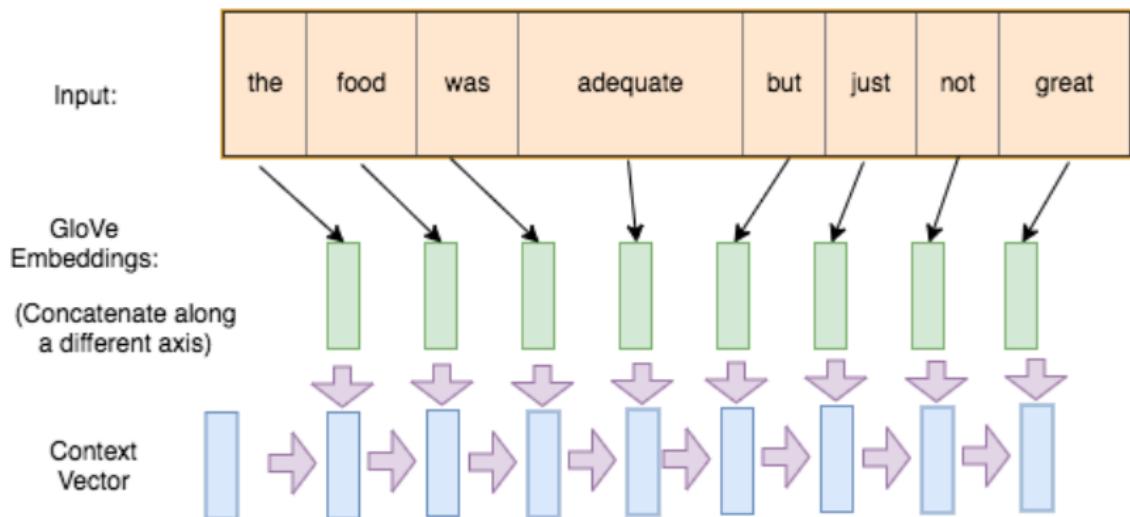
Compute the first hidden state based on the initial hidden state, and the input (the one-hot vector x_1 of the **first word**).

RNN: Continue Updating Hidden State



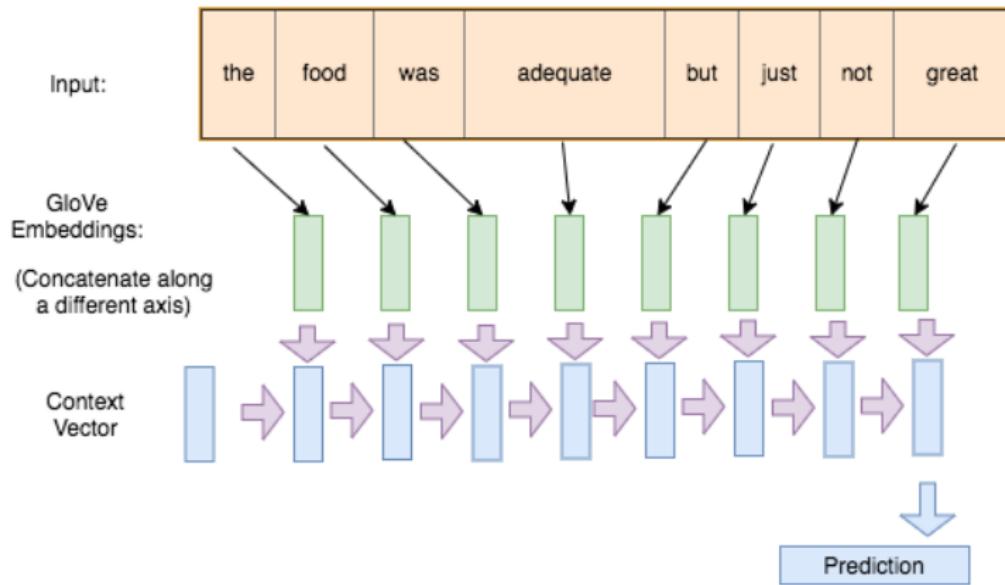
Update the hidden state based on the subsequent inputs. Note that we are using the **same weights** to perform the update each time.

RNN: Last Hidden State



Continue updating the hidden state until we run out of words in our sentence.

RNN: Compute Prediction



Use the **last hidden state** as input to a prediction network, usually a MLP.

Alternative: take the max-pool and average-pool over all computed hidden states. (Why?)

Next Time

- ▶ We'll talk more about RNNs next week.
- ▶ It turns out that this simple "vanilla" RNN is difficult to train due to **vanishing and exploding gradients**.
 - ▶ Explaining and addressing this issue will take a while
 - ▶ However, it will also help us understand the motivation between convolutional architectures like Residual Networks, which use skip connections

For now, you should be able to:

- ▶ Understand how the simple RNN examples work
- ▶ Understand the computation graph of the simple RNNs
- ▶ Be able to compute gradients for very simple RNN's (like the summation example)
- ▶ Have a rough idea of how RNNs can be used to for sentiment analysis of short text