CSC413 Neural Networks and Deep Learning

Lecture 8: Recurrent Neural Networks

March 5/7, 2024

Table of Contents

Lecture Plan

This week, we'll switch gears and talk about working with sequences via Recurrent Neural Networks

Section 1

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

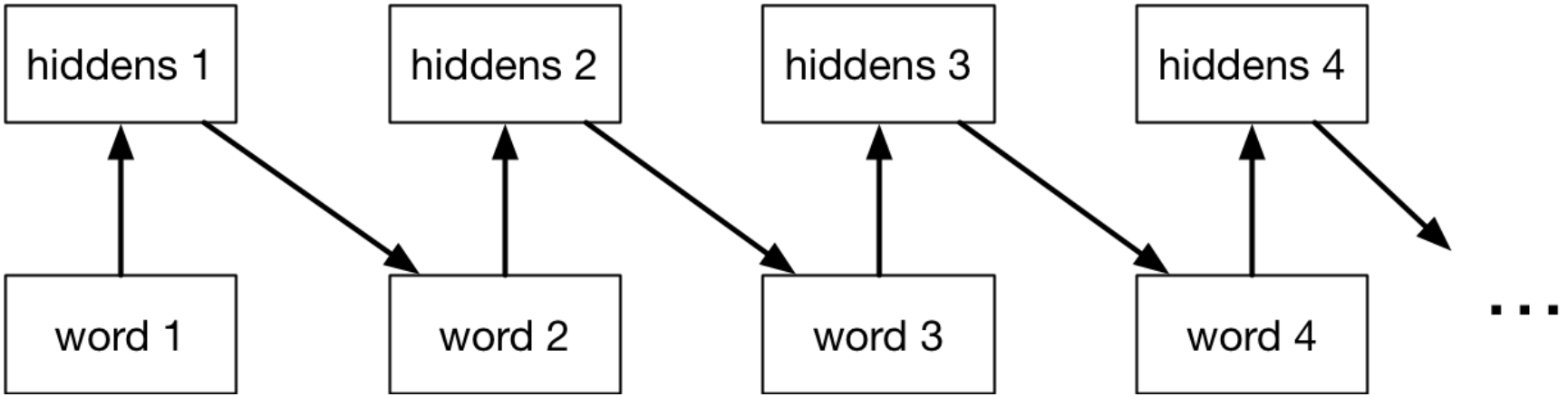
This last example is a **sequence-to-sequence prediction** task, because both inputs and outputs are sequences.

Language Model

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

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

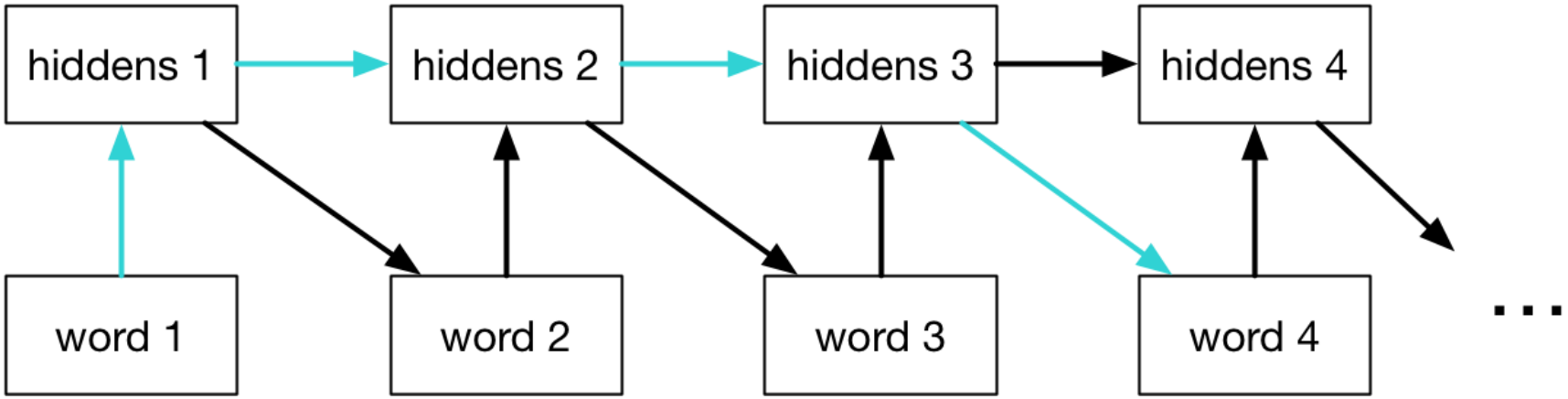
This means the model is **memoryless**, so they can only use information from their immediate context (in this figure, context length $= 1$):



Recurrent Neural Network

But sometimes long-distance context can be important.

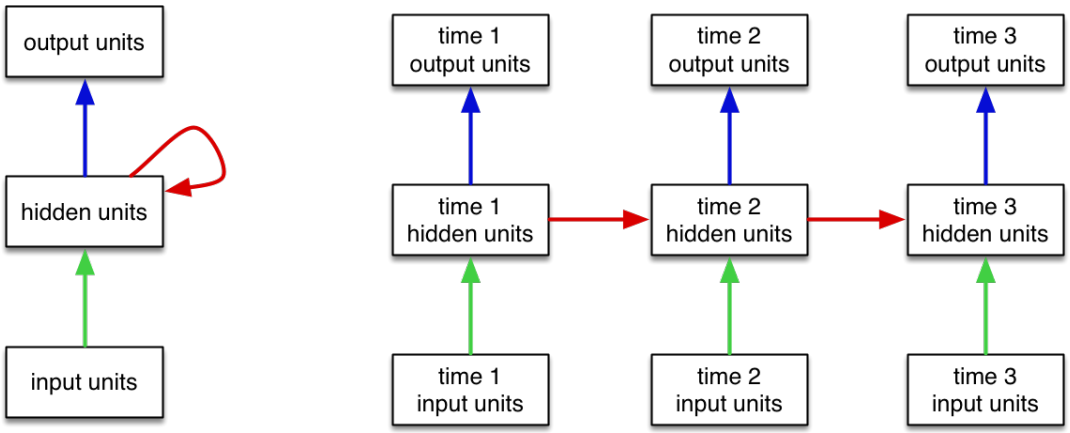
If we add connections between the hidden units, it becomes a **recurrent neural network (RNN)**. Having a memory lets an RNN use longer-term dependencies:



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

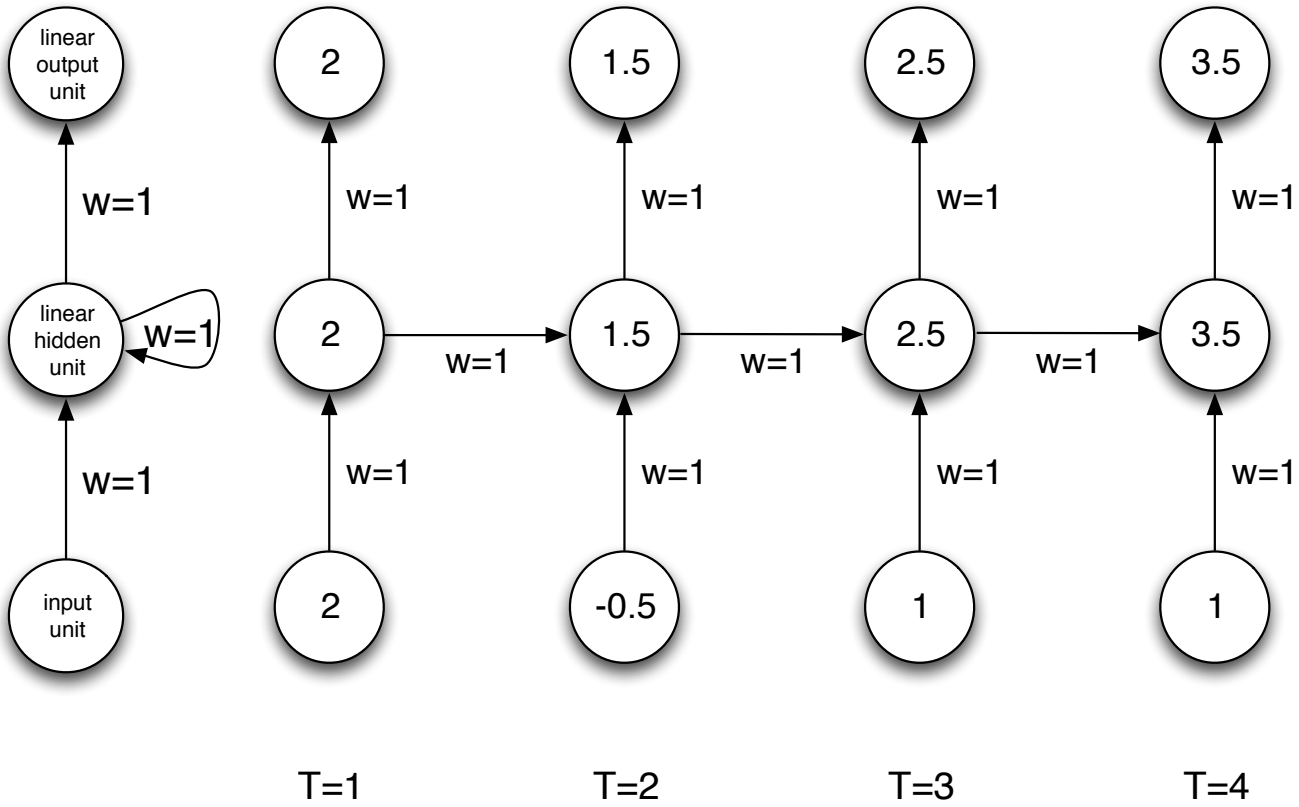


Simple RNNs

Let's go through a few examples of very simple RNNs to understand how RNNs compute predictions.

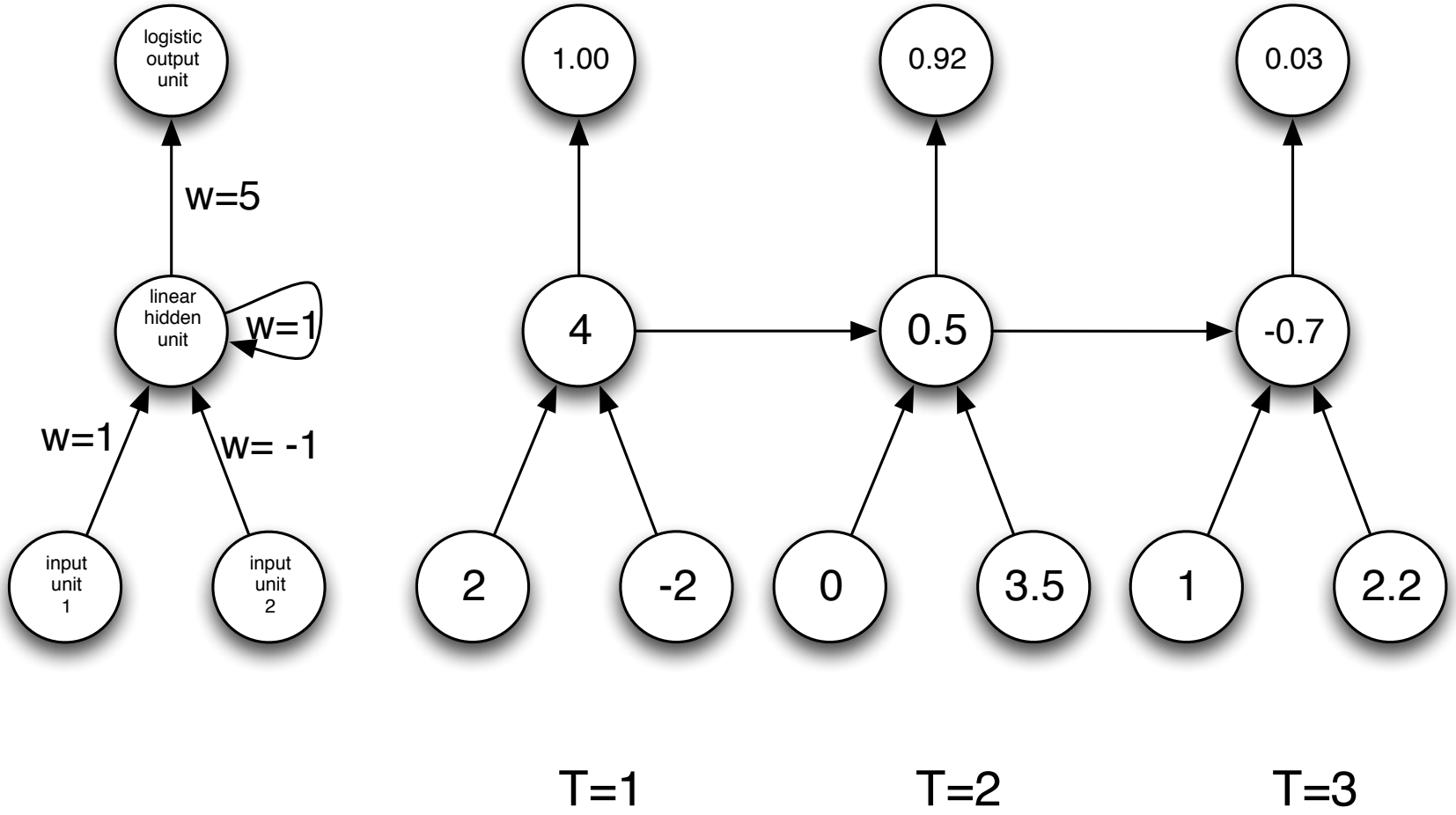
Simple RNN Example: Sum

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	$\longrightarrow$				
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.

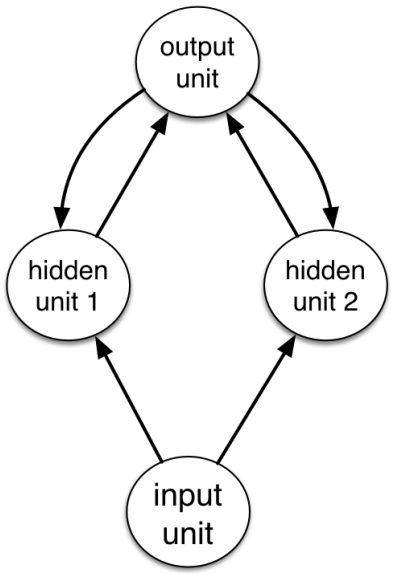
Parity is a classic example of a problem that's hard to solve with a shallow feed-forward net, but easy to solve with an RNN.

Parity: RNN Approach

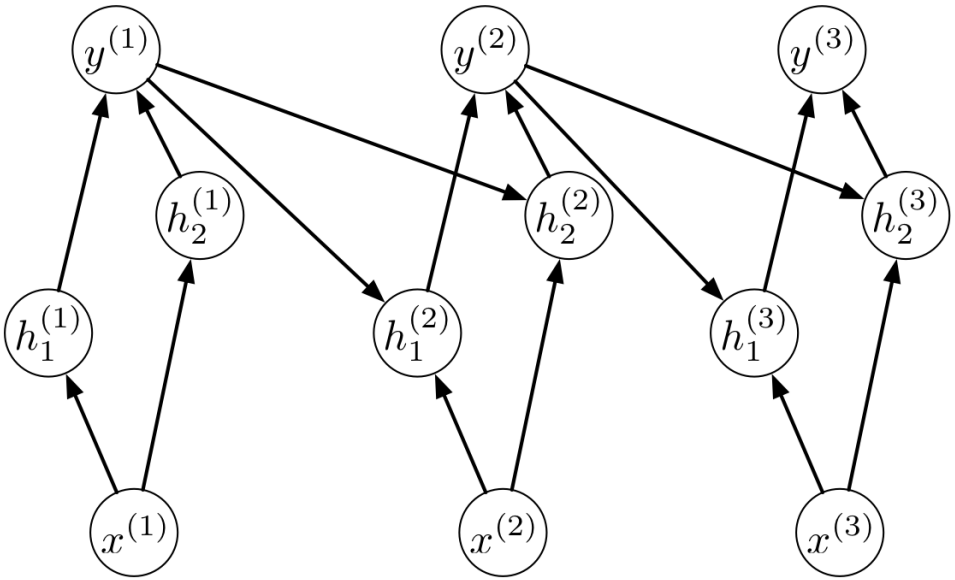
Let's find weights and biases for the RNN, so that it computes the parity. All hidden and output units are **binary threshold units** ($h(x) = 1$ if $x > 0$ and $h(x) = 0$ otherise).

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:

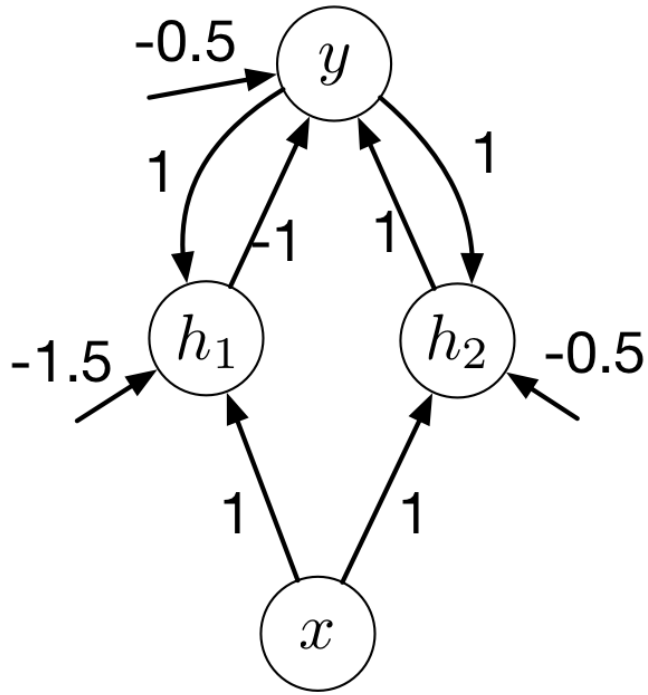
$y^{(t-1)}$	$x^{(t)}$	$y^{(t)}$
0	0	0
0	1	1
1	0	1
1	1	0

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.
- Note that a XOR b = (a OR b) - (a AND b)

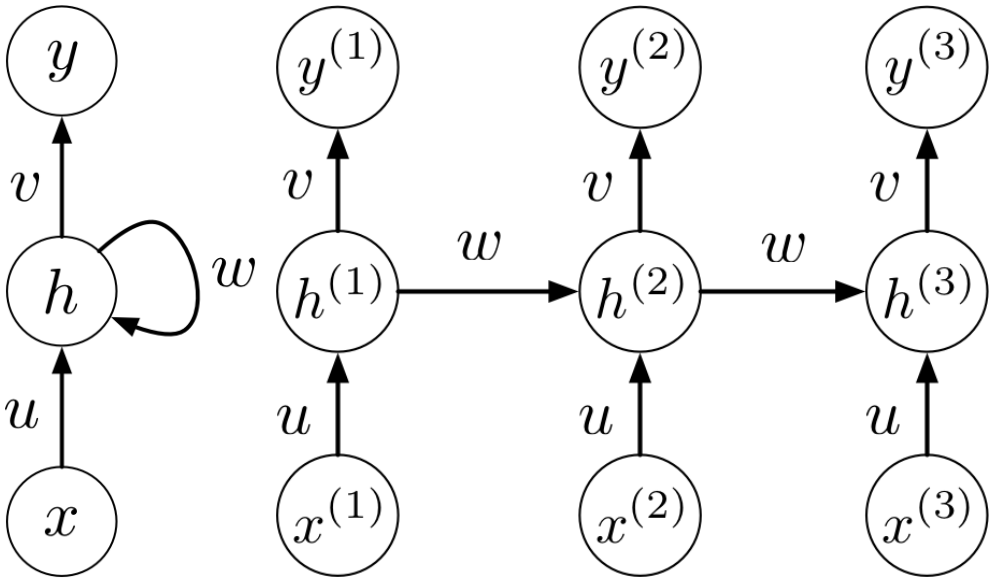
$y^{(t-1)}$	$x^{(t)}$	$h_1^{(t)}$	$h_2^{(t)}$	$y^{(t)}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



Back Propagation Through Time

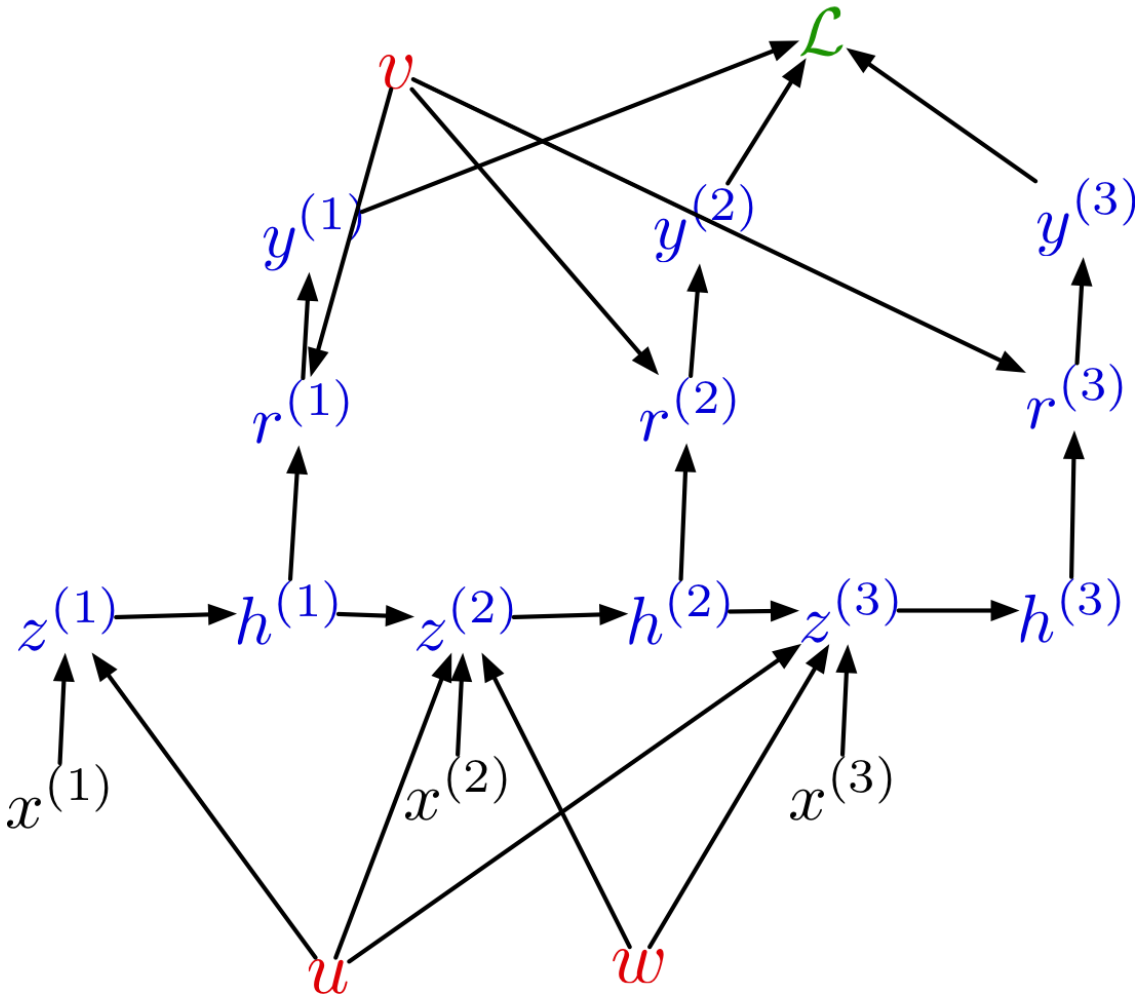
As you can guess, we don't usually set RNN weights by hand. Instead, we learn them using backprop.

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.



What can RNNs compute?

In 2014, Google researchers built an encoder-decoder RNN that learns to execute simple Python programs, one character at a time!
https://arxiv.org/abs/1410.4615

Input:
```
j=8584
for x in range(8):
  j+=920
b=(1500+j)
print((b+7567))
```
Target: 25011.

Input:
```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else
    5308)
```
Target: 1218.

Example training inputs

Input:
```
vqppkn
sqdvfljmnc
y2vxdddsepnimcbvubkomhrpliibtwztbljipcc
```
Target: hkhpg

A training input with characters scrambled

What can RNNs compute?

RNNs are good at learning complex syntactic structures: generate Algebraic Geometry LaTex source files that almost compiles:

For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compariciply in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ??. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example ??. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma ??. Namely, by Lemma ?? we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ *(by the formal open covering* X *and a single map* $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ *over* U *compatible with the complex*

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ *is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If* T *is surjective we may assume that* T *is connected with residue fields of* S*. Moreover there exists a closed subspace* $Z \subset X$ *of* X *where* U *in* X' *is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem*

(1) f *is locally of finite type. Since* $S = \mathrm{Spec}(R)$ *and* $Y = \mathrm{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{\mathcal{X},\ldots,0}$.

Lemma 0.2. *Let* X *be a locally Noetherian scheme over* S, $E = \mathcal{F}_{X/S}$. *Set* $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. *Since* $\mathcal{I}^n \subset \mathcal{I}^n$ *are nonzero over* $i_0 \leq \mathfrak{p}$ *is a subset of* $\mathcal{J}_{n,0} \circ \overline{A}_2$ *works.*

Lemma 0.3. *In Situation* ??. *Hence we may assume* $\mathfrak{q}' = 0$.

Proof. We will use the property we see that $\mathfrak{p}$ is the mext functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F-algebra where δ_{n+1} is a scheme over S. □

Section 2

Sentiment Analysis with Recurrent Neural Networks

RNN for language modelling

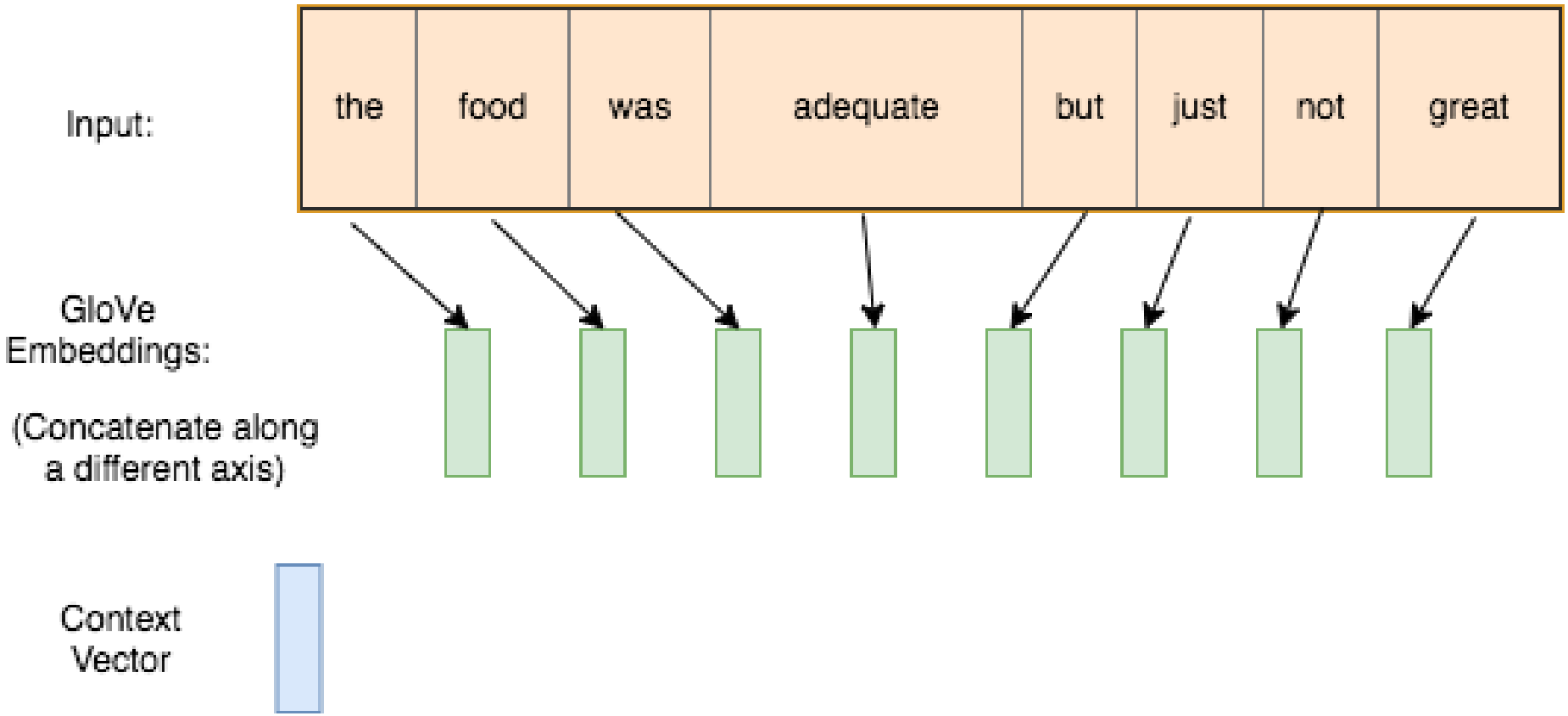
Usually, the sequence of inputs x_t will be **vectors**. The hidden states h_t are also vectors.

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

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

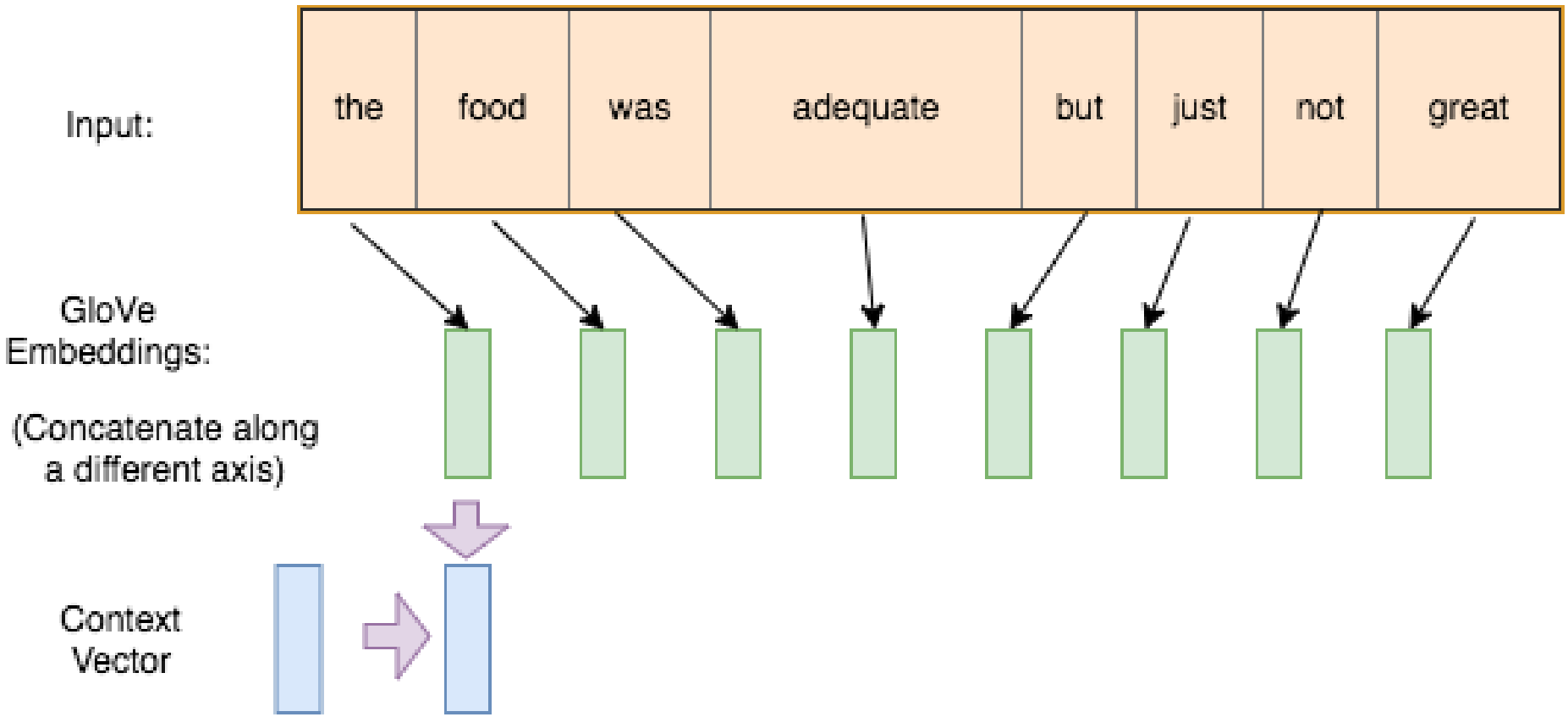
As usual, start with the forward pass...

RNN: Initial Hidden State



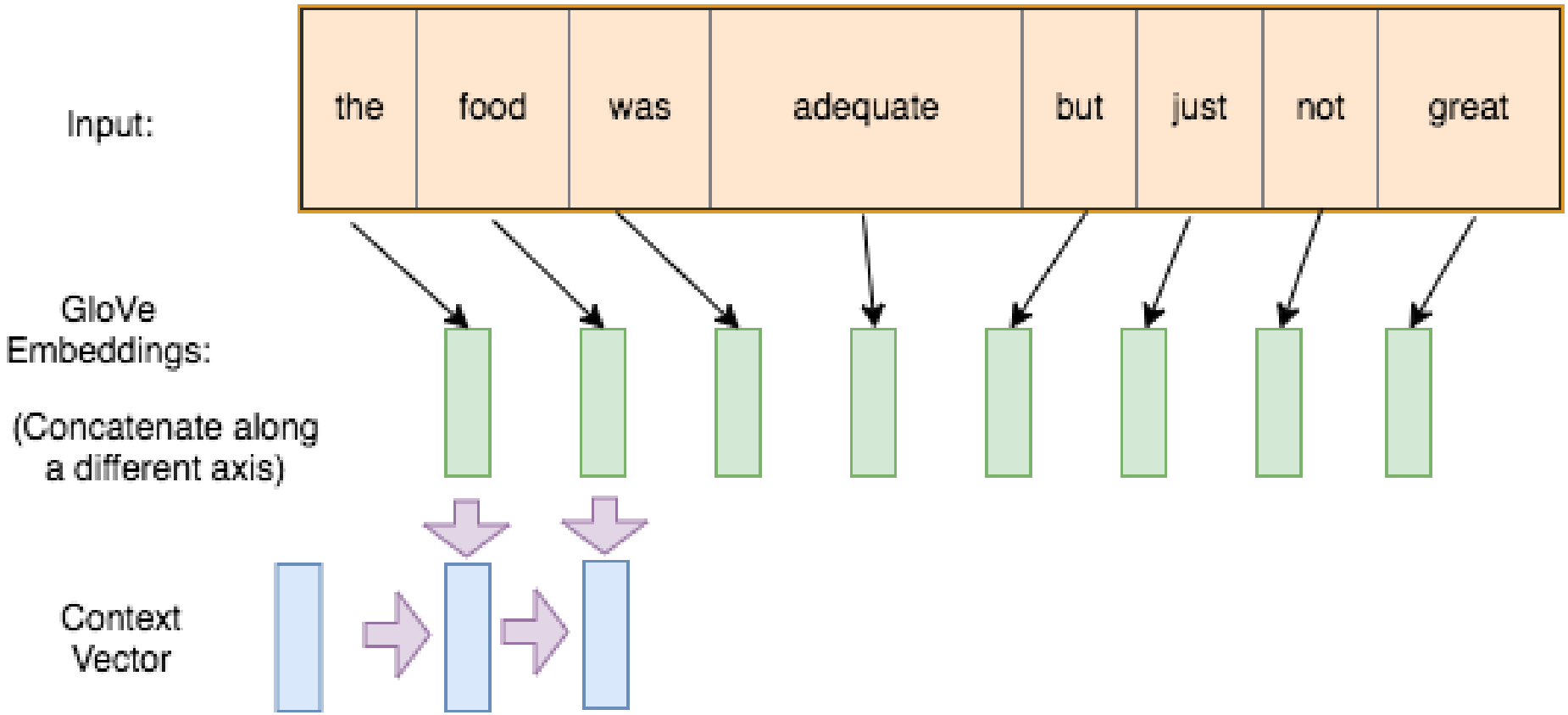
Start with an initial **hidden state** with a blank slate (can be a vector of all zeros, or a parameter that we train)

RNN: Update Hidden State



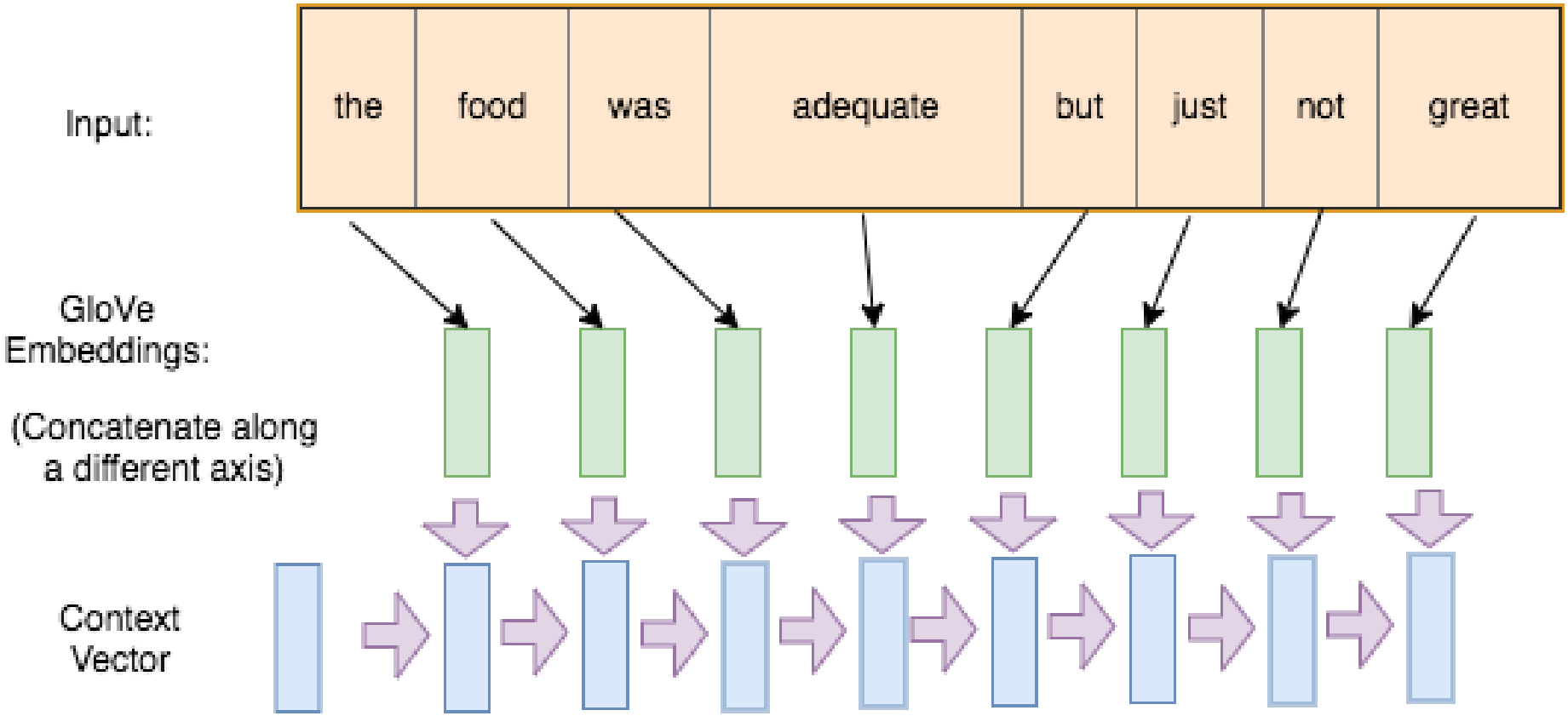
Compute the first hidden state (context vector) based on the initial hidden state, and the input (the one-hot vector $\mathbf{x}_1$ of the **first word**).

RNN: Continue Updating Hidden State



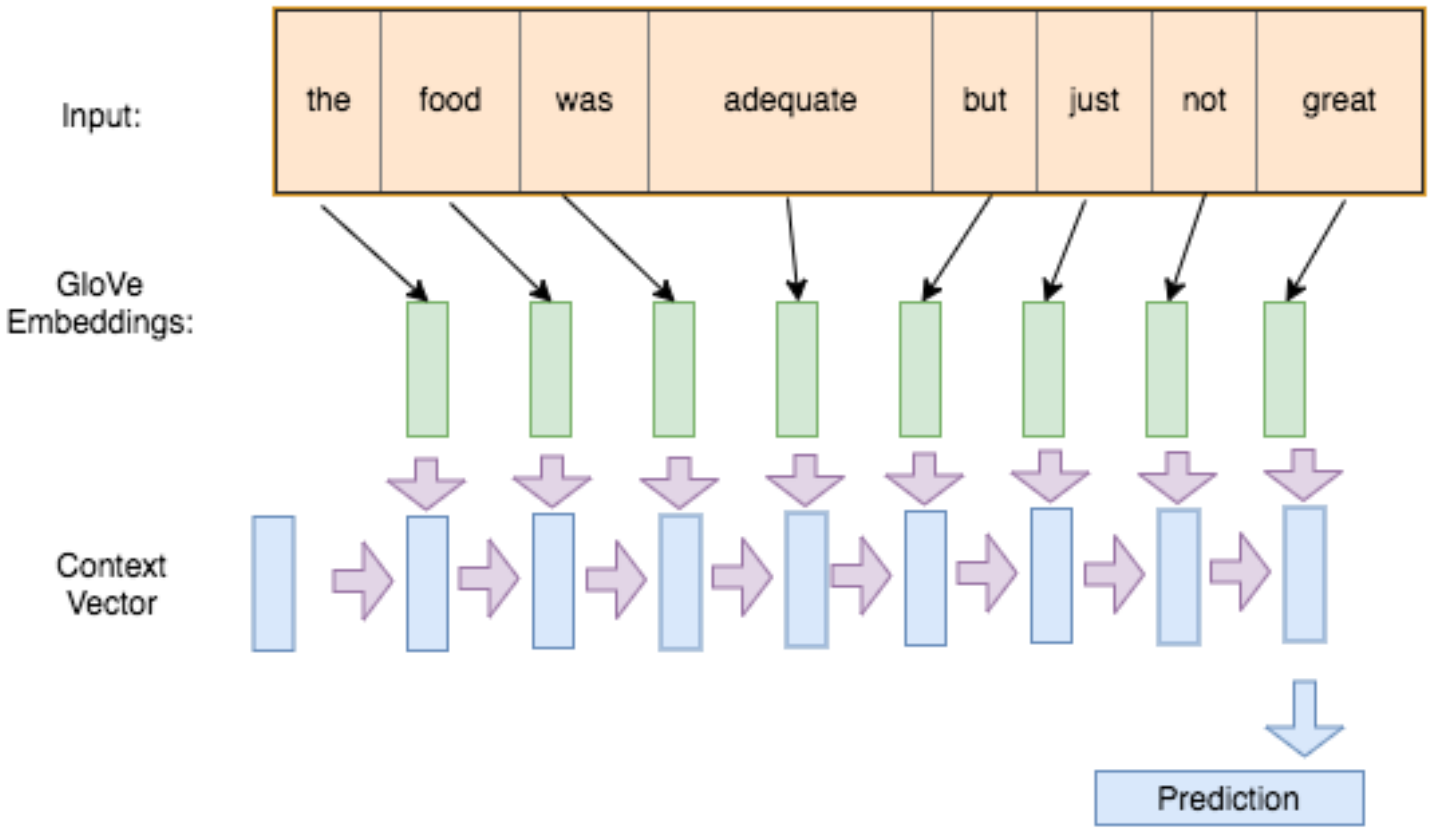
Update the hidden state based on the subsequent inputs. Note that we are using the **same weights** to perform the update each time.

RNN: Last Hidden State



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

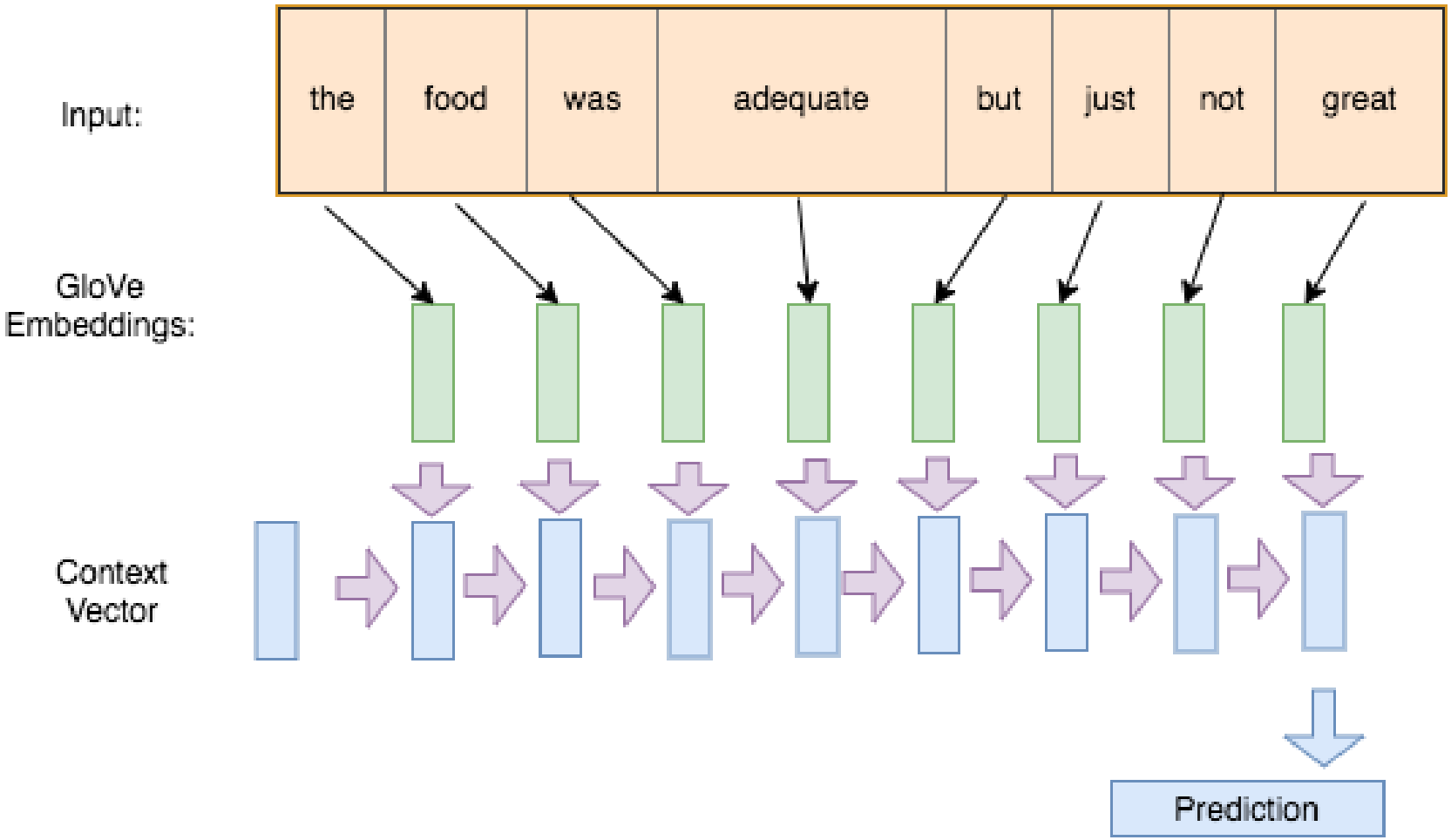
RNN: Compute Prediction



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?)

Sequence Classification



Sentiment140 Data

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

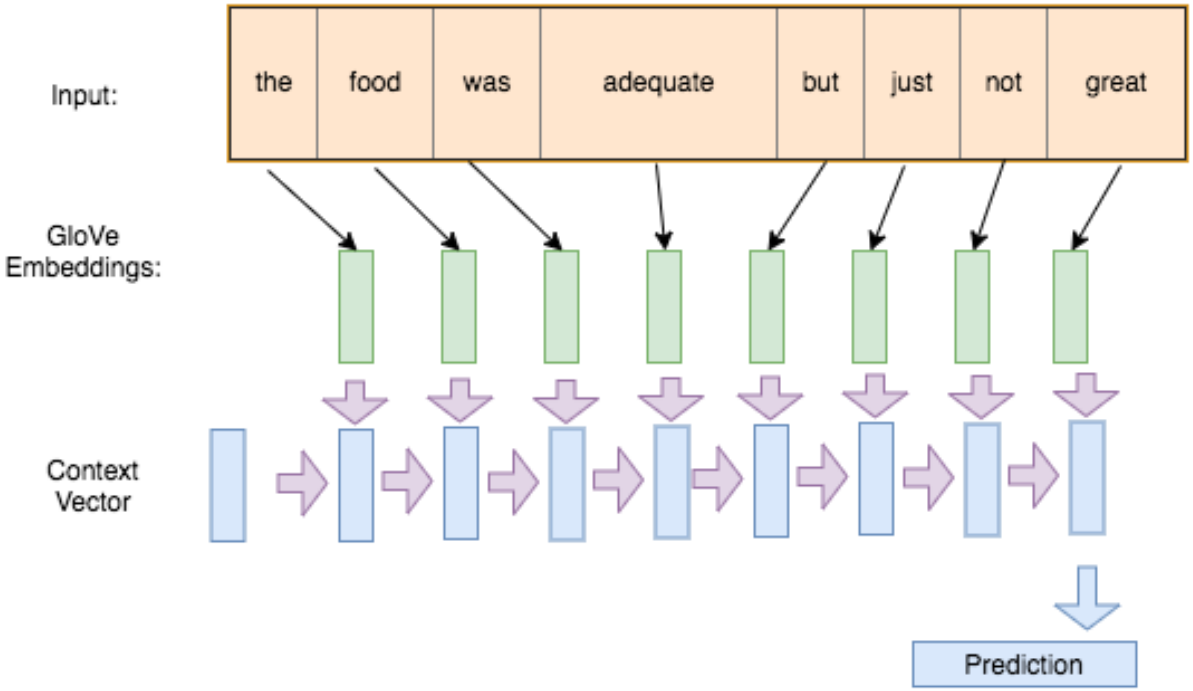
Input: Tweet (sequence of words/characters)

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)

Video Demo

Key Takeaways

You should be able to understand...

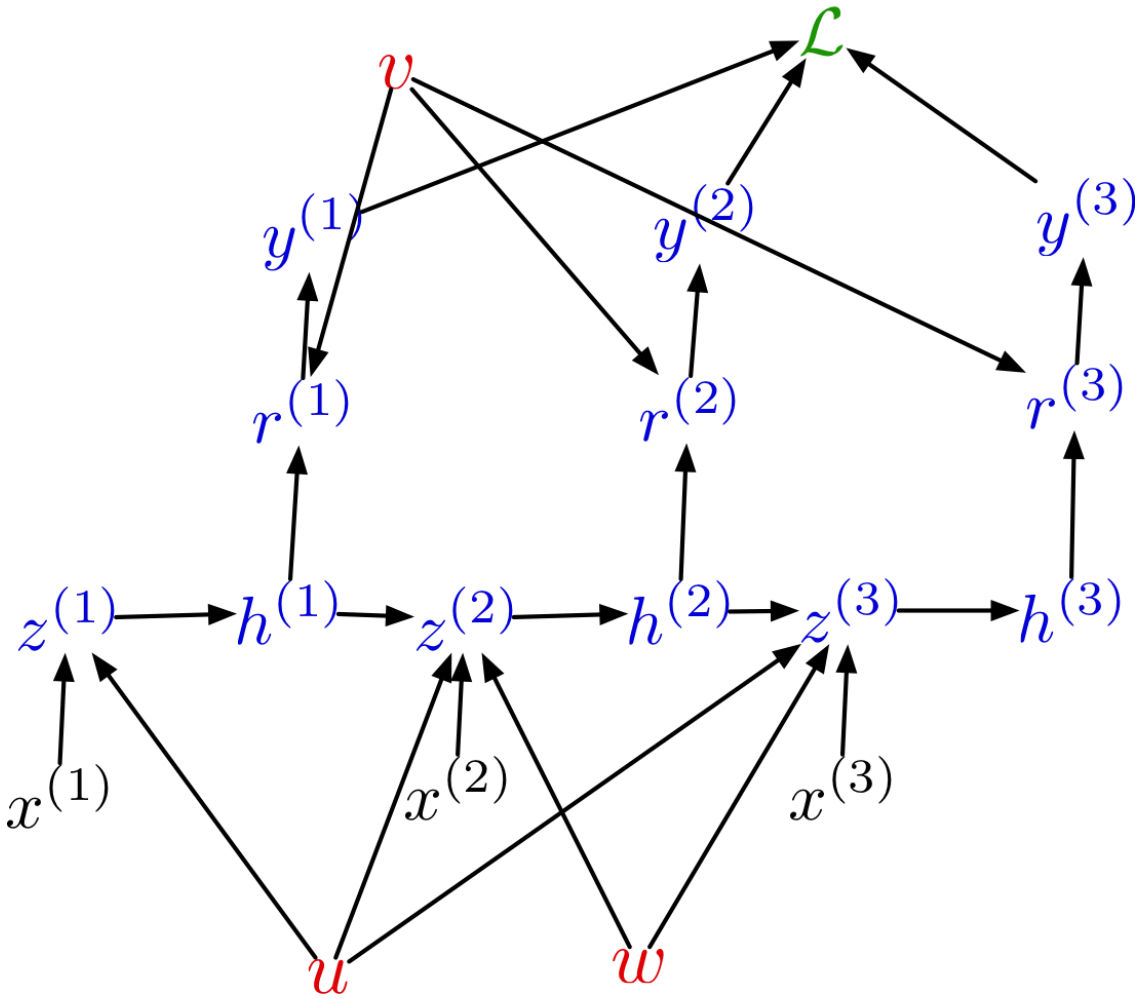
- why we want to use RNNs rather than CNN/MLP
- why/how GloVe embeddings are used in RNNs
- what the hidden state computations depend on (not the exact computation, but the dependencies)
- which weights are shared and which weights are not
- why batching is trickier when training an RNN (compared to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

Section 3

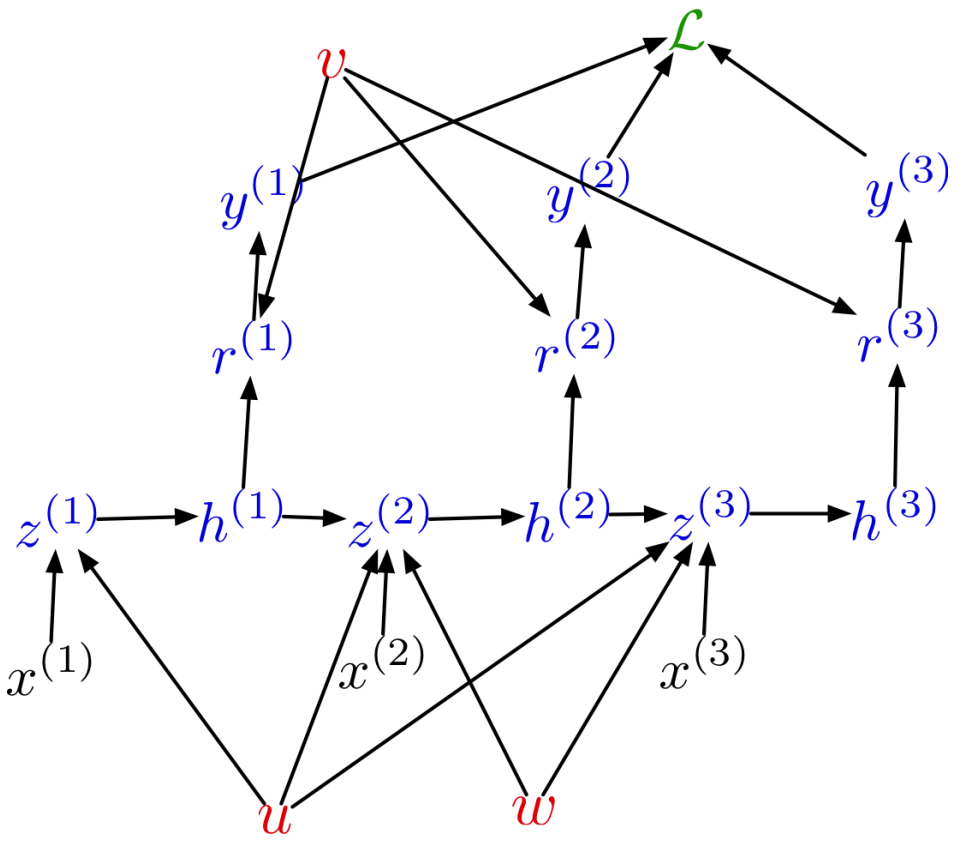
Gradient Explosion and Vanishing

RNN Gradients

Recall the unrolled computation graph for a small RNN:



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:

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

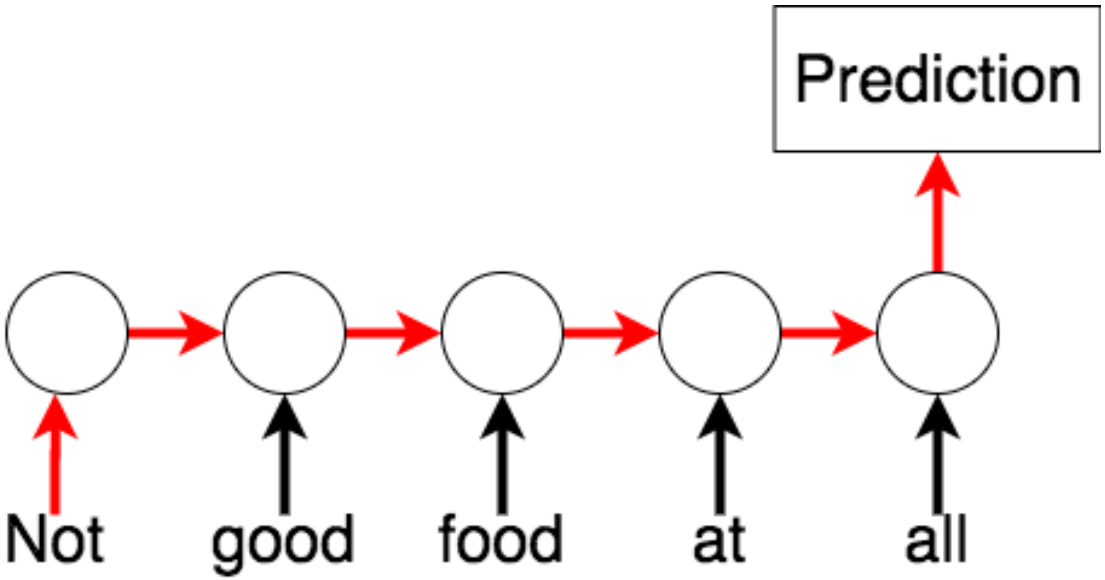
$$\overline{v} = \sum_t \overline{r^{(t)}}\, h^{(t)}$$

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

Key idea: multivariate chain rule!

Gradient Explosion and Vanishing

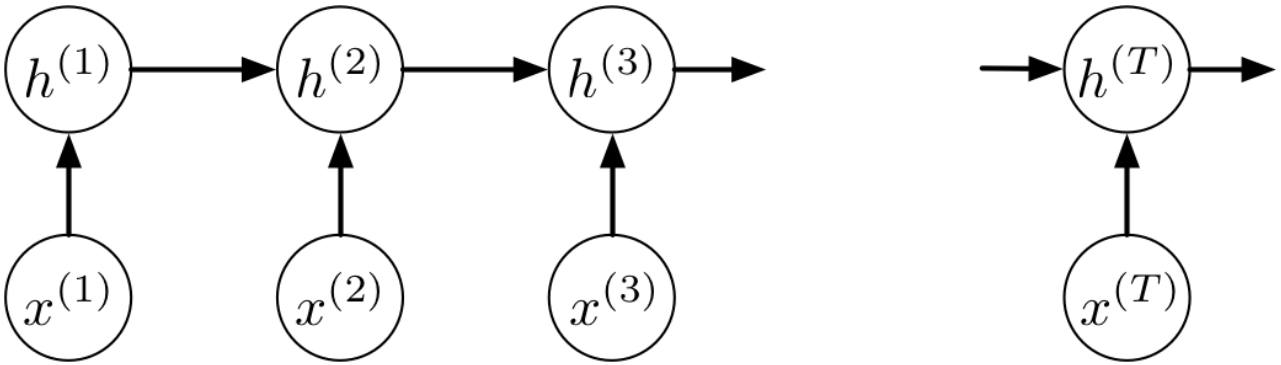
The longer your sequence, the longer gap the time step between when we see potentially important information and when we need it:



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 some simplifying assumptions:

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

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.

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.

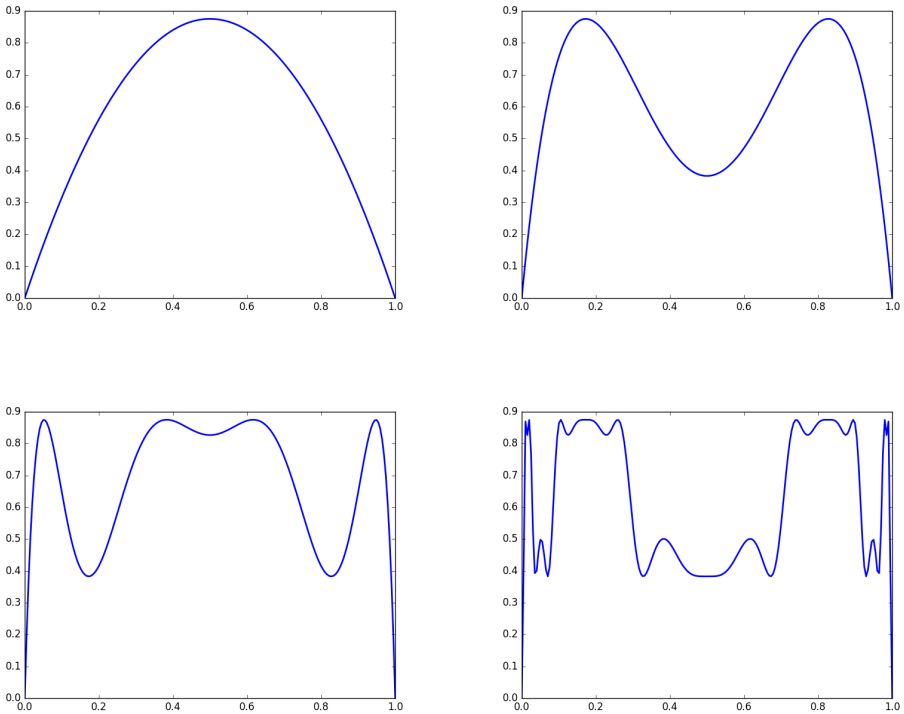
Each hidden layer computes some function of previous hidden layer and the current input: $\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)})$

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

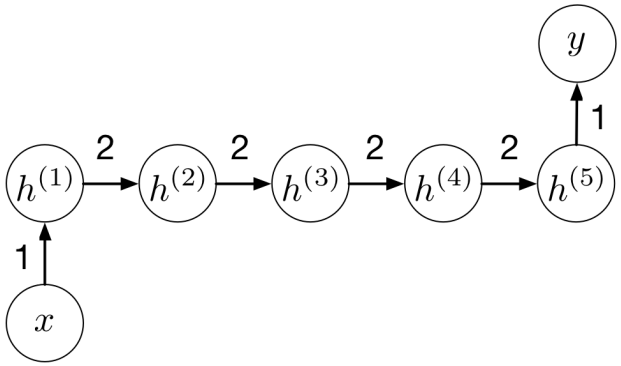
We get complicated behaviour from iterated functions. Consider $f(x) = 3.5x(1-x)$



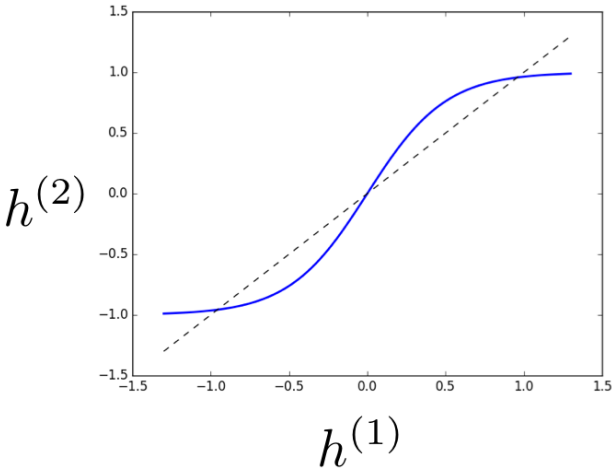
Note that the function values gravitate towards **fixed points**, and that the derivatives becomes either **very large** or **very small**.

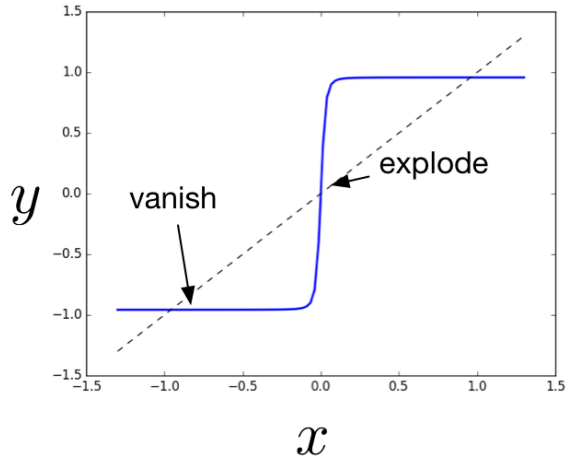
RNN with tanh activation

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



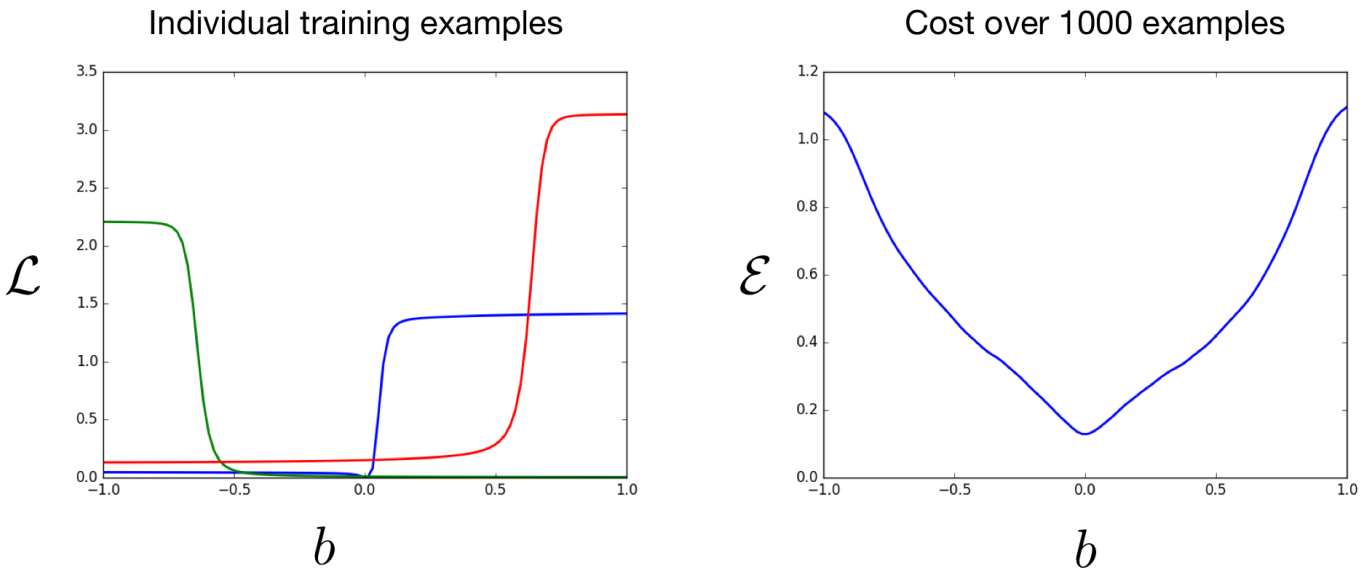
The function computed by the network:





Cliffs

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

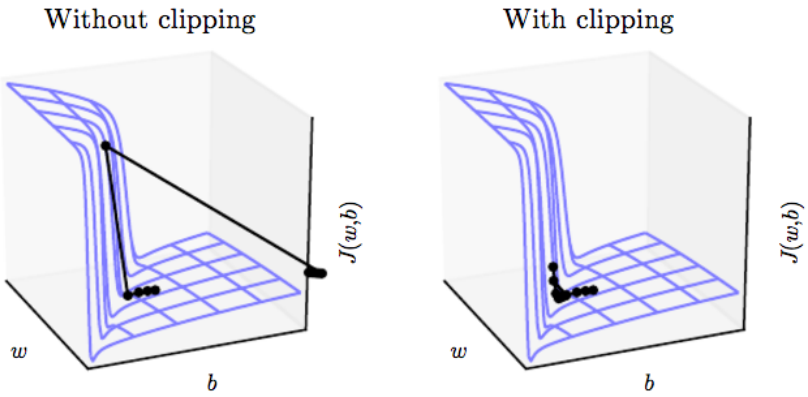


Generally, the gradient will explode on some inputs and vanish on others. In expectation, the cost may be fairly smooth.

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

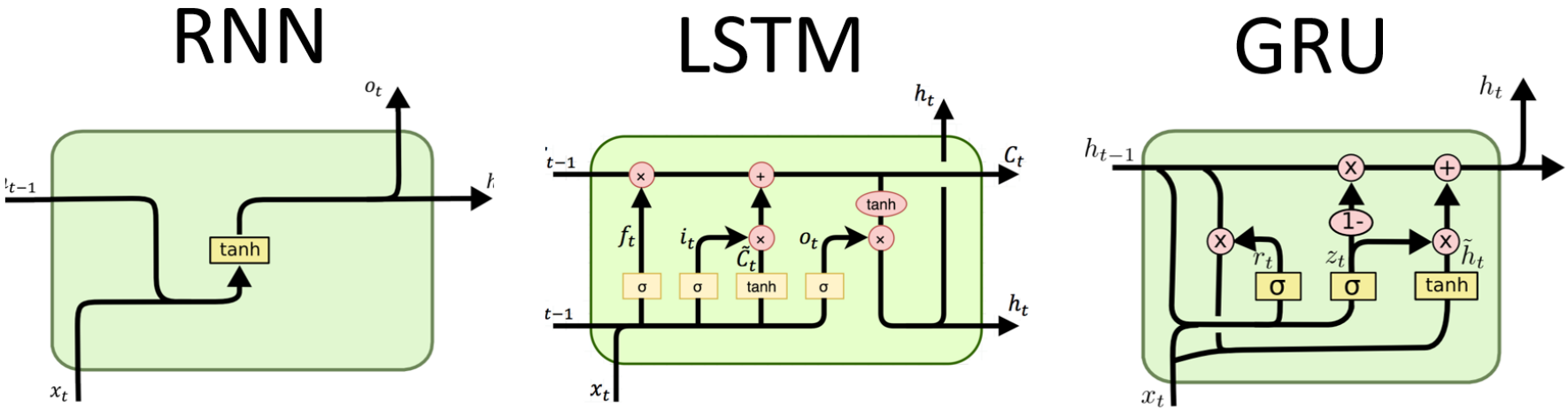
Hidden units are a kind of memory. Their default behaviour should be to **keep their previous value**.

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.

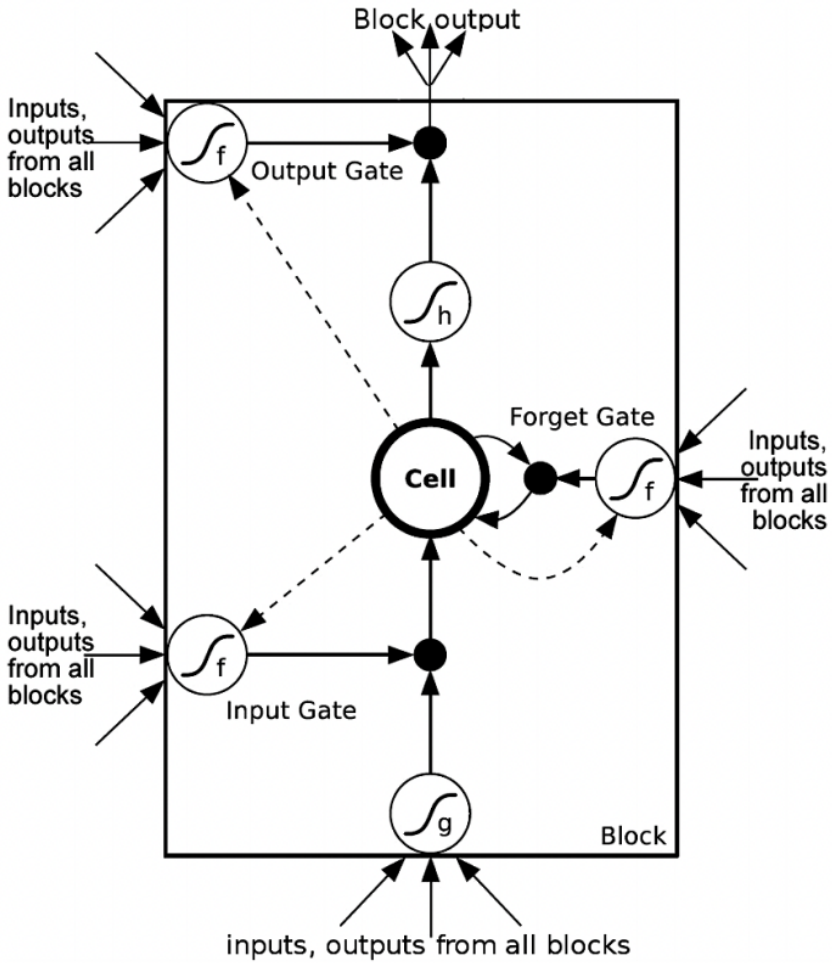
This initialization allows learning much longer-term dependencies than "vanilla" RNNs

Long-Term Short Term Memory

Change the **architecture** of the recurrent neural network by replacing each single unit in an RNN by a "memory block":



LSTM



$$c_{t+1} = c_t \cdot \text{forget gate} + \text{new input} \cdot \text{input gate}$$

- $i = 0, f = 1 \Rightarrow$ remember the previous value
- $i = 1, f = 1 \Rightarrow$ add to the previous value
- $i = 0, f = 0 \Rightarrow$ erase the value
- $i = 1, f = 0 \Rightarrow$ overwrite the value

Setting $i = 0, f = 1$ gives the reasonable "default" behavior of just remembering things.

LSTM Math

In each step, we have a vector of memory cells $\mathbf{c}$, a vector of hidden units $\mathbf{h}$ and a vector of input, output, and forget gates $\mathbf{i}$, $\mathbf{o}$ and $\mathbf{f}$.

There's a full set of connections from all the inputs and hiddens to the inputs and all of the gates:

$$\begin{pmatrix} \mathbf{i_t} \\ \mathbf{f_t} \\ \mathbf{o_t} \\ \mathbf{g_t} \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} \mathbf{W} \begin{pmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{pmatrix}$$

$$\mathbf{c_t} = \mathbf{f_t} \circ \mathbf{c_{t-1}} + \mathbf{i_t} \circ \mathbf{g_t}$$

$$\mathbf{h_t} = \mathbf{o_t} \circ tanh(\mathbf{c_t})$$

Exercise: show that if $\mathbf{f}_{t+1} = 1$, $\mathbf{i}_{t+1} = 0$, and $\mathbf{o}_t = 0$, the gradient of the memory cell gets passed through unmodified, i.e. $\bar{\mathbf{c}}_t = \bar{\mathbf{c}}_{t+1}$

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 mathematics behind why gating works

Section 4

Text Generation with RNN

RNN Hidden States

RNN For Prediction:

- Process tokens one at a time
- Hidden state is a representation of **all the tokens read thus far**

RNN Hidden States

RNN For Prediction:

- Process tokens one at a time
- Hidden state is a representation of **all the tokens read thus far**

RNN For Generation:

- Generate tokens one at a time
- Hidden state is a representation of **all the tokens to be generated**

RNN hidden state updates

RNN For Prediction:

- Update hidden state with new input (token)
- Get prediction (e.g. distribution over possible labels)

RNN hidden state updates

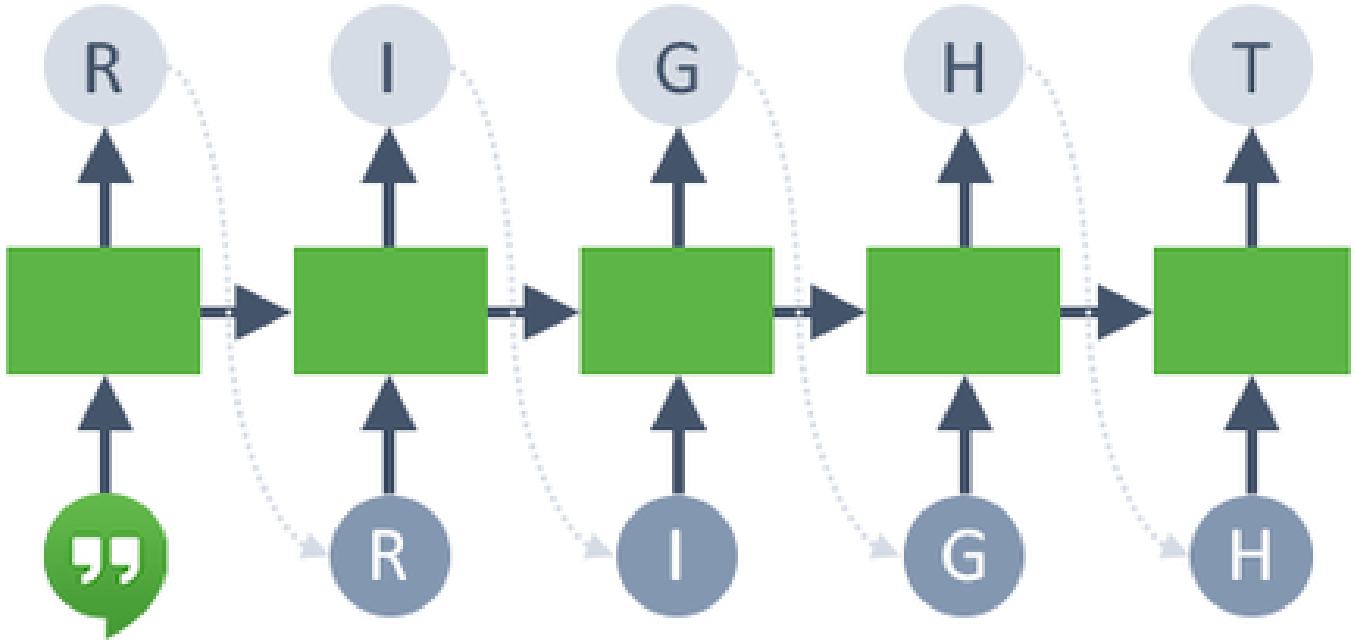
RNN For Prediction:

- Update hidden state with new input (token)
- Get prediction (e.g. distribution over possible labels)

RNN For Generation:

- Get prediction distribution of next token
- Generate a token from the distribution
- Update the hidden state with new token

Text Generation Diagram



- Get prediction distribution of next token
- Generate a token from the distribution
- Update the hidden state with new token:

Test Time Behaviour of Generative RNN

Unlike other models we discussed so far, the training time behaviour of Generative RNNs will be **different** from the test time behaviour

Test time behaviour:

- At each time step:
  - Obtain a **distribution** over possible next tokens
  - Sample a token from that distribution
  - Update the hidden state based on the sample token

Training Time Behaviour of Generative RNN

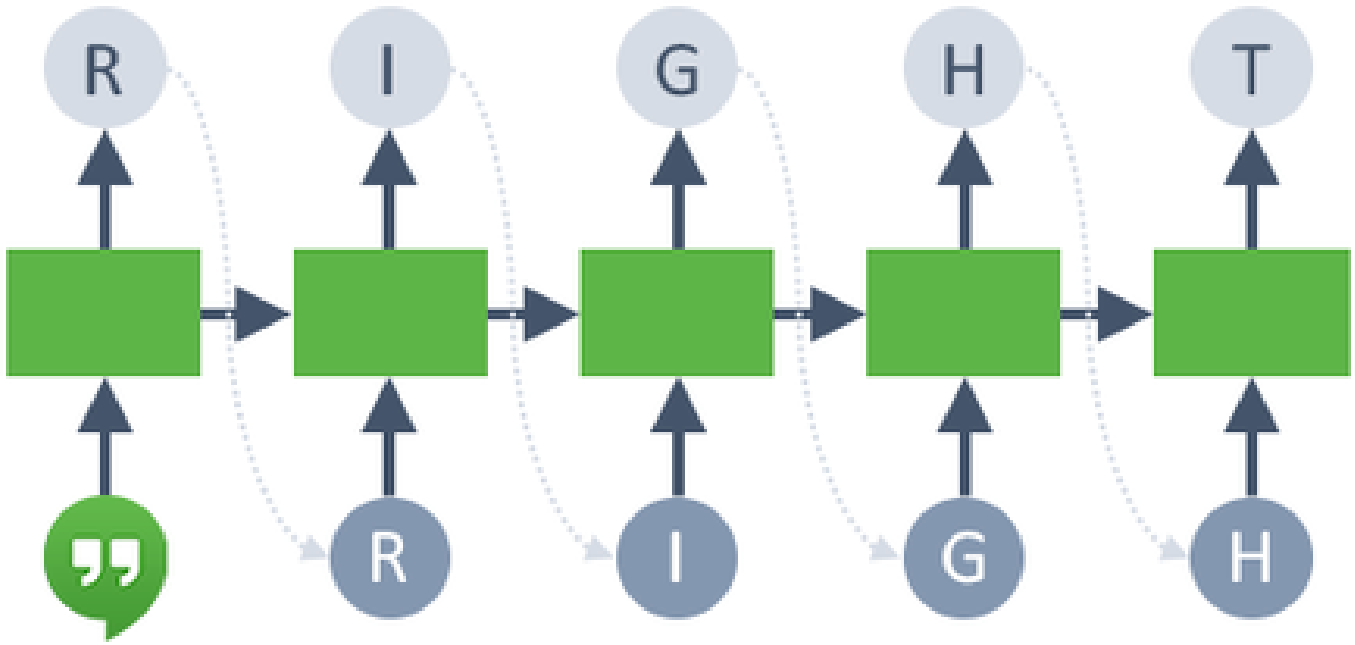
During training, we try to get the RNN to generate one particular sequence in the training set:

- At each time step:
  - Obtain a **distribution** over possible next tokens
  - Compare this with the *actual* next token

Q1: What kind of a problem is this? (regression or classification?)

Q2: What loss function should we use during training?

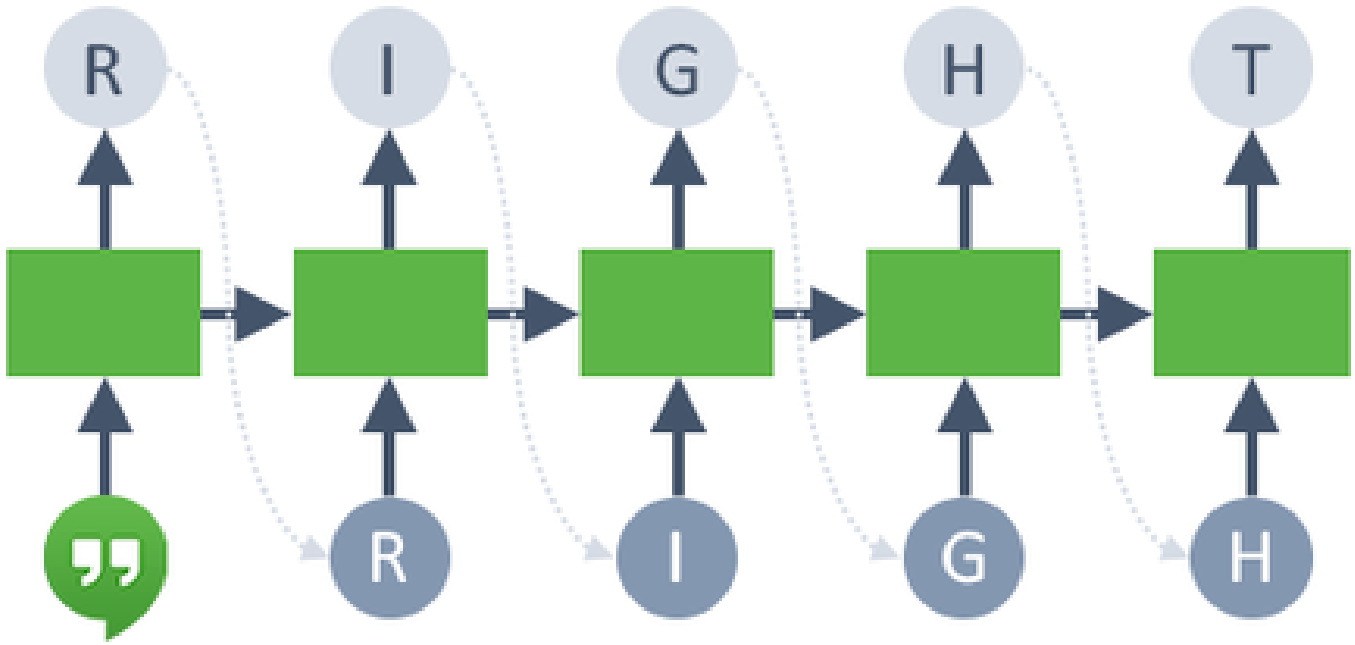
Text Generation: Step 1



First classification problem:

- Start with an initial hidden state
- Update the hidden state with a "<BOS>" (beginning of string) token, so that the hidden state becomes meaningful (not just zeros)
- Get the distribution over the first character
- Compute the cross-entropy loss against the ground truth (R)

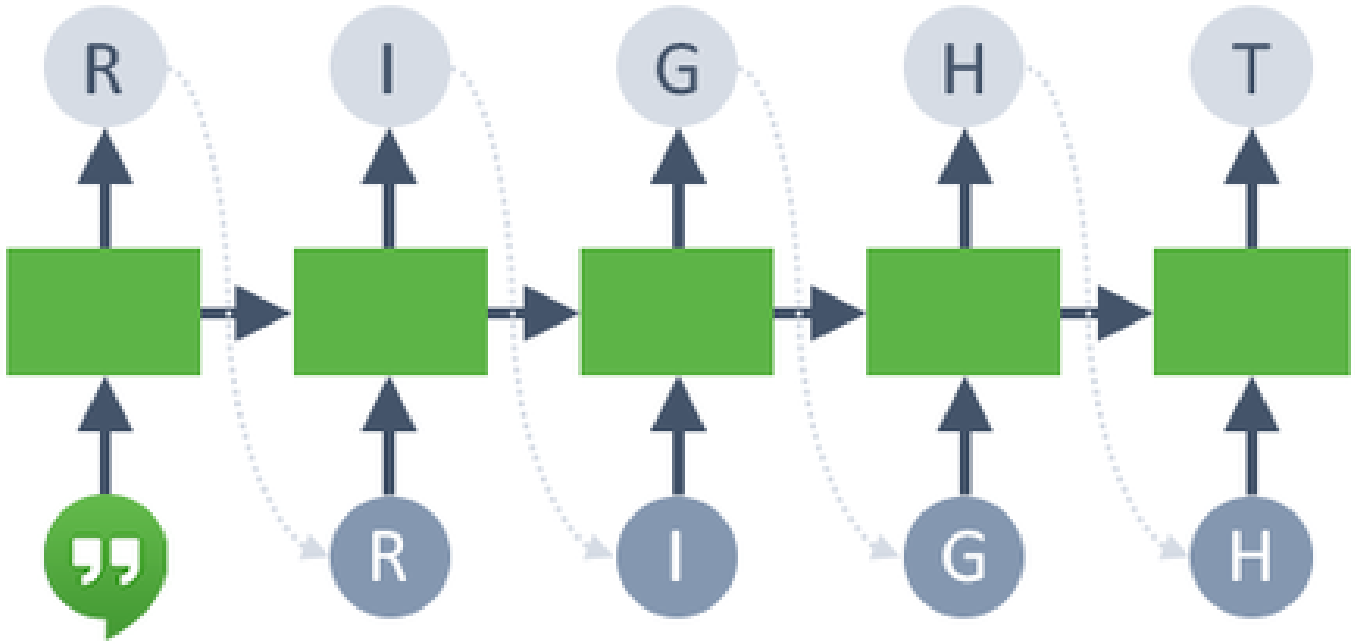
Text Generation with Teaching Forcing



Second classification problem:

- Update the hidden state with the **ground truth** token (R) regardless of the prediction from the previous step
  - This technique is called **teaching forcing**
- Get the distribution over the second character
- Compute the cross-entropy loss against the ground truth (I)

Text Generation: Later Steps



Continue until we get to the "<EOS>" (end of string) token

Some Remaining Challenges

- Vocabularies can be very large once you include people, places, etc.
- It's computationally difficult to predict distributions over millions of words.
- How do we deal with words we haven't seen before?
- In some languages (e.g. German), it's hard to define what should be considered a word.

Character vs word-level

Another approach is to model text *one character at a time*

This solves the problem of what to do about previously unseen words.

Note that long-term memory is essential at the character level!

Section 5

Sequence-to-Sequence Architecture

Neural Machine Translation

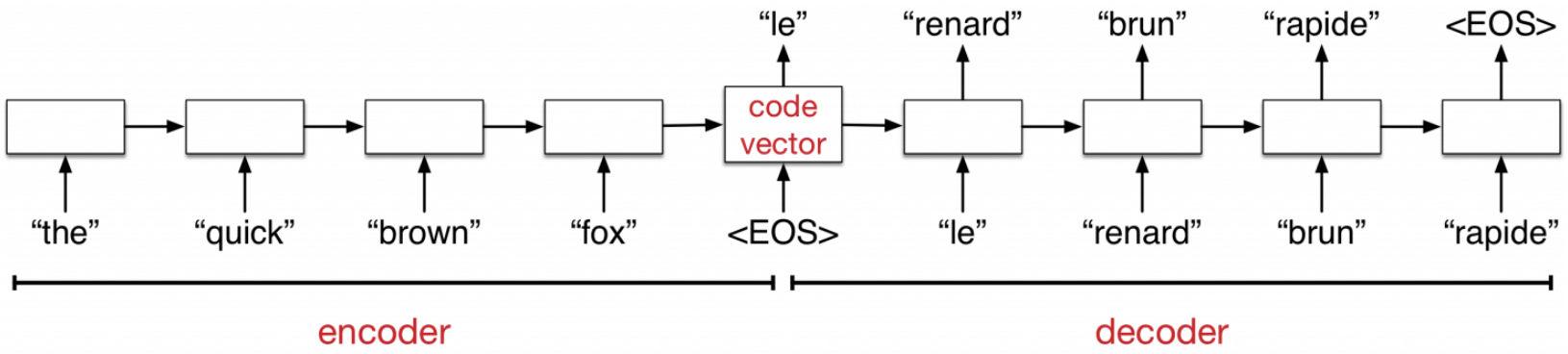
Say we want to translate, e.g. English to French sentences.

We have pairs of translated sentences to train on.

Here, both the inputs and outputs are sequences!

What can we do?

Sequence-to-sequence architecture



The network first reads and memorizes the sentences.

When it sees the "end token", it starts outputting the translation.

The "encoder" and "decoder" are two different networks with different weights.